3 books in 1

Effective C++ Digital Collection

140 Ways to Improve Your Programming

Scott Meyers

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Effective C++ Digital Collection

140 Ways to Improve Your Programming

Scott Meyers

ADDISON–WESLEY

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales

(800) 382-3419

corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales

international@pearson.com

Visit us on the Web: informit.com/aw

Copyright © 2012 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-13-297919-1

ISBN-10: 0-13-297919-5

Second release, September 2012

Note from the Publisher

The *Effective C++ Digital Collection* includes three bestselling C++ eBooks:

- Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition
- More Effective C++: 35 New Ways to Improve Your Programs and Designs
- Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library

By combining these seminal works into one eBook collection, you have easy access to the pragmatic, down-to-earth advice and proven wisdom that Scott Meyers is known for. This collection is essential reading for anyone working with C++ and STL.

To simplify access, we've appended "A" to pages of *Effective* C++, "B" to pages of *More Effective* C++, and ”C” to pages of *Effective STL*. This enabled us to produce a single, comprehensive table of contents and dedicated indexes so that you can easily link to the topics you want and navigate between the books. We hope you find this collection useful!

—The editorial and production teams at Addison-Wesley

Table of Contents

Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition

- Introduction

Chapter 1. Accustoming Yourself to C++

- Item 1: View C++ as a federation of languages
- Item 2: Prefer consts, enums, and inlines to #defines
- Item 3: Use const whenever possible
- Item 4: Make sure that objects are initialized before they're used

Chapter 2. Constructors, Destructors, and Assignment Operators

- Item 5: Know what functions C++ silently writes and calls
- Item 6: Explicitly disallow the use of compiler-generated functions you do not want
- Item 7: Declare destructors virtual in polymorphic base classes
- Item 8: Prevent exceptions from leaving destructors
- Item 9: Never call virtual functions during construction or destruction
- Item 10: Have assignment operators return a reference to *this
- Item 11: Handle assignment to self in operator=
- Item 12: Copy all parts of an object

Chapter 3. Resource Management

- Item 13: Use objects to manage resources.
- Item 14: Think carefully about copying behavior in resource-managing classes.
- Item 15: Provide access to raw resources in resource-managing classes.
- Item 16: Use the same form in corresponding uses of new and delete.
- Item 17: Store newed objects in smart pointers in standalone statements.

Chapter 4. Designs and Declarations

Item 18: Make interfaces easy to use correctly and hard to use incorrectly
Item 19: Treat class design as type design
Item 20: Prefer pass-by-reference-to-const to pass-by-value
Item 21: Don't try to return a reference when you must return an object
Item 22: Declare data members private
Item 23: Prefer non-member non-friend functions to member functions
Item 24: Declare non-member functions when type conversions should apply to all parameters
Item 25: Consider support for a non-throwing swap

Chapter 5. Implementations

Item 26: Postpone variable definitions as long as possible.
Item 27: Minimize casting.
Item 28: Avoid returning “handles” to object internals.
Item 29: Strive for exception-safe code.
Item 30: Understand the ins and outs of inlining.
Item 31: Minimize compilation dependencies between files

Chapter 6. Inheritance and Object-Oriented Design

Item 32: Make sure public inheritance models “is-a.”
Item 33: Avoid hiding inherited names
Item 34: Differentiate between inheritance of interface and inheritance of implementation
Item 35: Consider alternatives to virtual functions
Item 36: Never redefine an inherited non-virtual function
Item 37: Never redefine a function's inherited default parameter value
Item 38: Model “has-a” or “is-implemented-in-terms-of” through composition
Item 39: Use private inheritance judiciously
Item 40: Use multiple inheritance judiciously

Chapter 7. Templates and Generic Programming

Item 41: Understand implicit interfaces and compile-time polymorphism

- Item 42: Understand the two meanings of typename
- Item 43: Know how to access names in templatized base classes
- Item 44: Factor parameter-independent code out of templates
- Item 45: Use member function templates to accept “all compatible types.”
- Item 46: Define non-member functions inside templates when type conversions are desired
- Item 47: Use traits classes for information about types
- Item 48: Be aware of template metaprogramming

Chapter 8. Customizing new and delete

- Item 49: Understand the behavior of the new-handler
- Item 50: Understand when it makes sense to replace new and delete
- Item 51: Adhere to convention when writing new and delete
- Item 52: Write placement delete if you write placement new

Chapter 9. Miscellany

- Item 53: Pay attention to compiler warnings
- Item 54: Familiarize yourself with the standard library, including TR1
- Item 55: Familiarize yourself with Boost.

Appendix A. Beyond Effective C++

Appendix B. Item Mappings Between Second and Third Editions

Index

More Effective C++ 35 New Ways to Improve Your Programs and Designs

Introduction

Basics

- Item 1: Distinguish between pointers and references.
- Item 2: Prefer C++-style casts.
- Item 3: Never treat arrays polymorphically.
- Item 4: Avoid gratuitous default constructors.

Operators

Item 5: Be wary of user-defined conversion functions.

Item 6: Distinguish between prefix and postfix forms of increment and decrement operators.

Item 7: Never overload `&&`, `||`, or `,`.

Item 8: Understand the different meanings of `new` and `delete`.

Exceptions

Item 9: Use destructors to prevent resource leaks.

Item 10: Prevent resource leaks in constructors.

Item 11: Prevent exceptions from leaving destructors.

Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function.

Item 13: Catch exceptions by reference.

Item 14: Use exception specifications judiciously.

Item 15: Understand the costs of exception handling.

Efficiency

Item 16: Remember the 80-20 rule.

Item 17: Consider using lazy evaluation.

Item 18: Amortize the cost of expected computations.

Item 19: Understand the origin of temporary objects.

Item 20: Facilitate the return value optimization.

Item 21: Overload to avoid implicit type conversions.

Item 22: Consider using *op*= instead of stand-alone *op*.

Item 23: Consider alternative libraries.

Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI.

Techniques

Item 25: Virtualizing constructors and non-member functions.

Item 26: Limiting the number of objects of a class.

Item 27: Requiring or prohibiting heap-based objects.

Item 28: Smart pointers.
Item 29: Reference counting.
Item 30: Proxy classes.
Item 31: Making functions virtual with respect to more than one object.

Miscellany

Item 32: Program in the future tense.
Item 33: Make non-leaf classes abstract.
Item 34: Understand how to combine C++ and C in the same program.
Item 35: Familiarize yourself with the language standard.

Recommended Reading

An `auto_ptr` Implementation

General Index

Index of Example Classes, Functions, and Templates

Effective STL 50 Specific Ways to Improve Your Use of the Standard Template Library

Introduction

Chapter 1. Containers

Item 1: Choose your containers with care
Item 2: Beware the illusion of container-independent code
Item 3: Make copying cheap and correct for objects in containers
Item 4: Call empty instead of checking size() against zero
Item 5: Prefer range member functions to their single-element counterparts
Item 6: Be alert for C++'s most vexing parse
Item 7: When using containers of newed pointers, remember to delete the pointers before the container is destroyed
Item 8: Never create containers of auto_ptrs
Item 9: Choose carefully among erasing options

Item 10: Be aware of allocator conventions and restrictions
Item 11: Understand the legitimate uses of custom allocators
Item 12: Have realistic expectations about the thread safety of STL containers
Chapter 2. vector and string
Item 13: Prefer vector and string to dynamically allocated arrays
Item 14: Use reserve to avoid unnecessary reallocations
Item 15: Be aware of variations in string implementations
Item 16: Know how to pass vector and string data to legacy APIs
Item 17: Use “the swap trick” to trim excess capacity
Item 18: Avoid using vector<bool>
Chapter 3. Associative Containers
Item 19: Understand the difference between equality and equivalence
Item 20: Specify comparison types for associative containers of pointers
Item 21: Always have comparison functions return false for equal values
Item 22: Avoid in-place key modification in set and multiset
Item 23: Consider replacing associative containers with sorted vectors
Item 24: Choose carefully between map::operator[] and map::insert when efficiency is important
Item 25: Familiarize yourself with the nonstandard hashed containers
Chapter 4. Iterators
Item 26: Prefer iterator to const_iterator, reverse_iterator, and const_reverse_iterator
Item 27: Use distance and advance to convert a container’s const_iterators to iterators
Item 28: Understand how to use a reverse_iterator’s base iterator
Item 29: Consider istreambuf_iterators for character-by-character input
Chapter 5. Algorithms
Item 30: Make sure destination ranges are big enough
Item 31: Know your sorting options

Item 32: Follow remove-like algorithms by erase if you really want to remove something
Item 33: Be wary of remove-like algorithms on containers of pointers
Item 34: Note which algorithms expect sorted ranges
Item 35: Implement simple case-insensitive string comparisons via mismatch or lexicographical_compare
Item 36: Understand the proper implementation of copy_if
Item 37: Use accumulate or for_each to summarize ranges

Chapter 6. Functors, Functor Classes, Functions, etc

Item 38: Design functor classes for pass-by-value
Item 39: Make predicates pure functions
Item 40: Make functor classes adaptable
Item 41: Understand the reasons for ptr_fun, mem_fun, and mem_fun_ref
Item 42: Make sure less<T> means operator<

Chapter 7. Programming with the STL

Item 43: Prefer algorithm calls to hand-written loops
Item 44: Prefer member functions to algorithms with the same names
Item 45: Distinguish among count, find, binary_search, lower_bound, upper_bound, and equal_range
Item 46: Consider function objects instead of functions as algorithm parameters
Item 47: Avoid producing write-only code
Item 48: Always #include the proper headers
Item 49: Learn to decipher STL-related compiler diagnostics
Item 50: Familiarize yourself with STL-related web sites

Bibliography

Appendix A. Locales and Case-Insensitive String Comparisons

Appendix B. Remarks on Microsoft's STL Platforms

Index

Effective C++

55 Specific Ways to Improve Your Programs and Designs, Third Edition

Scott Meyers

ADDISON–WESLEY

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the U.S., please contact:

International Sales
international@pearson.com

Visit us on the Web: www.awprofessional.com

Library of Congress Control Number: 2005924388

Copyright © 2005 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
One Lake Street
Upper Saddle River, NJ 07458

ISBN 0-321-33487-6

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.

First printing, May 2005

For Nancy,
without whom nothing
would be much worth doing

Wisdom and beauty form a very rare combination.

—Petronius Arbiter
Satyricon, XCIV

And in memory of Persephone, 1995–2004

Praise for *Effective C++, Third Edition*

“Scott Meyers' book, *Effective C++, Third Edition,* is distilled programming experience — experience that you would otherwise have to learn the hard way. This book is a great resource that I recommend to everybody who writes C++ professionally.”

— Peter Dulimov, ME, Engineer, Ranges and Assessing Unit, NAVSYSCOM, Australia

“The third edition is still the best book on how to put all of the pieces of C++ together in an efficient, cohesive manner. If you claim to be a C++ programmer, you must read this book.”

— Eric Nagler, Consultant, Instructor, and author of Learning C++

“The first edition of this book ranks among the small (very small) number of books that I credit with significantly elevating my skills as a 'professional' software devel-oper. Like the others, it was practical and easy to read, but loaded with important advice. *Effective C++, Third Edition*, continues that tradition. C++ is a very powerful programming language. If C gives you enough rope to hang yourself, C++ is a hardware store with lots of helpful people ready to tie knots for you. Mastering the points discussed in this book will definitely increase your ability to effectively use C++ and reduce your stress level.”

— Jack W. Reeves, Chief Executive Officer, Bleading Edge Software Technologies

“Every new developer joining my team has one assignment — to read this book.”

— Michael Lanzetta, Senior Software Engineer

“I read the first edition of *Effective C++* about nine years ago, and it immediately became my favorite book on C++. In my opinion, *Effective C++, Third Edition*, remains a must-read today for anyone who wishes to program effectively in C++. We would live in a better world if C++

programmers had to read this book before writing their first line of professional C++ code.”

— Danny Rabbani, Software Development Engineer

“I encountered the first edition of Scott Meyers' *Effective C++* as a struggling programmer in the trenches, trying to get better at what I was doing. What a lifesaver! I found Meyers' advice was practical, useful, and effective, fulfilling the promise of the title 100 percent. The third edition brings the practical realities of using C++ in serious development projects right up to date, adding chapters on the language's very latest issues and features. I was delighted to still find myself learning something interesting and new from the latest edition of a book I already thought I knew well.”

— Michael Topic, Technical Program Manager

“From Scott Meyers, the guru of C++, this is the definitive guide for anyone who wants to use C++ safely and effectively, or is transitioning from any other OO language to C++. This book has valuable information presented in a clear, concise, entertaining, and insightful manner.”

— Siddhartha Karan Singh, Software Developer

“This should be the second book on C++ that any developer should read, after a general introductory text. It goes beyond the *how* and *what* of C++ to address the *why* and *wherefore*. It helped me go from knowing the syntax to understanding the philosophy of C++ programming.”

— Timothy Knox, Software Developer

“This is a fantastic update of a classic C++ text. Meyers covers a lot of new ground in this volume, and every serious C++ programmer should have a copy of this new edition.”

— Jeffrey Somers, Game Programmer

“*Effective C++, Third Edition*, covers the things you should be doing when writing code and does a terrific job of explaining why those things are important. Think of it as best practices for writing C++.”

— Jeff Scherpelz, Software Development Engineer

“As C++ embraces change, Scott Meyers' *Effective C++, Third Edition,* soars to remain in perfect lock-step with the language. There are many fine introductory books on C++, but exactly one *second* book stands head and shoulders above the rest, and you're holding it. With Scott guiding the way, prepare to do some soaring of your own!”

— Leor Zolman, C++ Trainer and Pundit, BD Software

“This book is a must-have for both C++ veterans and newbies. After you have finished reading it, it will not collect dust on your bookshelf — you will refer to it all the time.”

— Sam Lee, Software Developer

“Reading this book transforms ordinary C++ programmers into expert C++ programmers, step-by-step, using 55 easy-to-read items, each describing one technique or tip.”

— Jeffrey D. Oldham, Ph.D., Software Engineer, Google

“Scott Meyers' *Effective C++* books have long been required reading for new and experienced C++ programmers alike. This new edition, incorporating almost a decade's worth of C++ language development, is his most content-packed book yet. He does not merely describe the problems inherent in the language, but instead he provides unambiguous and easy-to-follow advice on how to avoid the pitfalls and write 'effective C++.' I expect every C++ programmer to have read it.”

— Philipp K. Janert, Ph.D., Software Development Manager

“Each previous edition of *Effective C++* has been the must-have book for developers who have used C++ for a few months or a few years, long enough to stumble into the traps latent in this rich language. In this third edition, Scott Meyers extensively refreshes his sound advice for the modern world of new language and library features and the programming styles that have evolved to use them. Scott's engaging writing style makes it easy to assimilate his guidelines on your way to becoming an effective C++ developer.”

— David Smallberg, Instructor, DevelopMentor; Lecturer, Computer Science, UCLA

“*Effective C++* has been completely updated for twenty-first-century C++ practice and can continue to claim to be the first *second* book for all C++ practitioners.”

— Matthew Wilson, Ph.D., author of Imperfect C++

Preface

I wrote the original edition of *Effective C++* in 1991. When the time came for a second edition in 1997, I updated the material in important ways, but, because I didn't want to confuse readers familiar with the first edition, I did my best to retain the existing structure: 48 of the original 50 Item titles remained essentially unchanged. If the book were a house, the second edition was the equivalent of freshening things up by replacing carpets, paint, and light fixtures.

For the third edition, I tore the place down to the studs. (There were times I wished I'd gone all the way to the foundation.) The world of C++ has undergone enormous change since 1991, and the goal of this book — to identify the most important C++ programming guidelines in a small, readable package — was no longer served by the Items I'd established nearly 15 years earlier. In 1991, it was reasonable to assume that C++ programmers came from a C background. Now, programmers moving to C++ are just as likely to come from Java or C#. In 1991, inheritance and object-oriented programming were new to most programmers. Now they're well-established concepts, and exceptions, templates, and generic programming are the areas where people need more guidance. In 1991, nobody had heard of design patterns. Now it's hard to discuss software systems without referring to them. In 1991, work had just begun on a formal standard for C++. Now that standard is eight years old, and work has begun on the next version.

To address these changes, I wiped the slate as clean as I could and asked myself, “What are the most important pieces of advice for practicing C++ programmers in 2005?” The result is the set of Items in this new edition. The book has new chapters on resource management and on programming with templates. In fact, template concerns are woven throughout the text, because they affect almost everything in C++. The book also includes new material on programming in the presence of exceptions, on applying design patterns, and on using the new TR1 library facilities. (TR1 is described in Item 54.) It acknowledges that techniques and approaches that work well in single-threaded systems may not be appropriate in multithreaded systems. Well over half the material in the book is new. However, most of the fundamental information in the second edition continues to be important, so I found a way to retain it in one form or another. (You'll find a mapping between the second and third edition Items in Appendix B.)

I've worked hard to make this book as good as I can, but I have no illusions that it's perfect. If you feel that some of the Items in this book are inappropriate as general advice; that there is a better way to accomplish a task examined in the book; or that one or more of the technical discussions is unclear, incomplete, or misleading, please tell me. If you find an error of any kind — technical, grammatical, typographical, *whatever* — please tell me that, too. I'll gladly add to the acknowledgments in later printings the name of the first person to bring each problem to my attention.

Even with the number of Items expanded to 55, the set of guidelines in this book is far from exhaustive. But coming up with good rules — ones that apply to almost all applications almost all the time — is harder than it might seem. If you have suggestions for additional guidelines, I would be delighted to hear about them.

I maintain a list of changes to this book since its first printing, including bug fixes, clarifications, and technical updates. The list is available at the *Effective C++ Errata* web page, http://aristeia.com/BookErrata/ec++3e-errata.html. If you'd like to be notified when I update the list, I encourage you to join my mailing list. I use it to make announcements likely to interest people who follow my professional work. For details, consult http://aristeia.com/MailingList/.

SCOTT DOUGLAS MEYERS
http://aristeia.com/
STAFFORD, OREGON
APRIL 2005

Acknowledgments

Effective C++ has existed for fifteen years, and I started learning C++ about three years before I wrote the book. The "*Effective C++* project" has thus been under development for nearly two decades. During that time, I have benefited from the insights, suggestions, corrections, and, occasionally, dumbfounded stares of hundreds (thousands?) of people. Each has helped improve *Effective C++*. I am grateful to them all.

I've given up trying to keep track of where I learned what, but one general source of information has helped me as long as I can remember: the Usenet C++ newsgroups, especially `comp.lang.c++.moderated` and `comp.std.c++`. Many of the Items in this book — perhaps most — have benefited from the vetting of technical ideas at which the participants in these newsgroups excel.

Regarding new material in the third edition, Steve Dewhurst worked with me to come up with an initial set of candidate Items. In Item 11, the idea of implementing `operator=` via copy-and-`swap` came from Herb Sutter's writings on the topic, e.g., Item 13 of his *Exceptional C++* (Addison-Wesley, 2000). RAII (see Item 13) is from Bjarne Stroustrup's *The C++ Programming Language* (Addison-Wesley, 2000). The idea behind Item 17 came from the "Best Practices" section of the Boost `shared_ptr` web page, http://boost.org/libs/smart_ptr/shared_ptr.htm#BestPractices and was refined by Item 21 of Herb Sutter's *More Exceptional C++* (Addison-Wesley, 2002). Item 29 was strongly influenced by Herb Sutter's extensive writings on the topic, e.g., Items 8-19 of *Exceptional C++*, Items 17–23 of *More Exceptional C++*, and Items 11–13 of *Exceptional C++ Style* (Addison-Wesley, 2005); David Abrahams helped me better understand the three exception safety guarantees. The NVI idiom in Item 35 is from Herb Sutter's column, "Virtuality," in the September 2001 *C/C++ Users Journal*. In that same Item, the Template Method and Strategy design patterns are from *Design Patterns* (Addison-Wesley, 1995) by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. The idea of using the NVI idiom in Item 37 came from Hendrik Schober. David Smallberg contributed the motivation for writing a custom `set` implementation in Item 38. Item 39's observation that the EBO generally isn't available under multiple inheritance is from David Vandevoorde's and Nicolai M. Josuttis' *C++ Templates* (Addison-Wesley, 2003). In Item 42, my initial understanding about `typename` came from Greg Comeau's C++ and C FAQ (http://www.comeaucomputing.com/techtalk/#typename), and Leor Zolman helped me realize that my understanding was incorrect. (My fault, not Greg's.) The essence of Item 46 is from Dan Saks' talk, “Making New Friends.” The idea at the end of Item 52 that if you declare one version of `operator new`, you should declare them all, is from Item 22 of Herb Sutter's *Exceptional C++ Style*. My understanding of the Boost review process (summarized in Item 55) was refined by David Abrahams.

Everything above corresponds to who or where *I* learned about something, not necessarily to who or where the thing was invented or first published.

My notes tell me that I also used information from Steve Clamage, Antoine Trux, Timothy Knox, and Mike Kaelbling, though, regrettably, the notes fail to tell me how or where.

Drafts of the first edition were reviewed by Tom Cargill, Glenn Carroll, Tony Davis, Brian Kernighan, Jak Kirman, Doug Lea, Moises Lejter, Eugene Santos, Jr., John Shewchuk, John Stasko, Bjarne Stroustrup, Barbara Tilly, and Nancy L. Urbano. I received suggestions for improvements that I was able to incorporate in later printings from Nancy L. Urbano, Chris Treichel, David Corbin, Paul Gibson, Steve Vinoski, Tom Cargill, Neil Rhodes, David Bern, Russ Williams, Robert Brazile, Doug Morgan, Uwe Steinmüller, Mark Somer, Doug Moore, David Smallberg, Seth Meltzer, Oleg Shteynbuk, David Papurt, Tony Hansen, Peter McCluskey, Stefan Kuhlins, David Braunegg, Paul Chisholm, Adam Zell, Clovis Tondo, Mike Kaelbling, Natraj Kini, Lars Nyman, Greg Lutz, Tim Johnson, John Lakos, Roger Scott, Scott Frohman, Alan Rooks, Robert Poor, Eric Nagler, Antoine Trux, Cade Roux, Chandrika Gokul, Randy Mangoba, and Glenn Teitelbaum.

Drafts of the second edition were reviewed by Derek Bosch, Tim Johnson, Brian Kernighan, Junichi Kimura, Scott Lewandowski, Laura Michaels, David Smallberg, Clovis Tondo, Chris Van Wyk, and Oleg Zabluda. Later printings benefited from comments from Daniel Steinberg, Arunprasad Marathe, Doug Stapp, Robert Hall, Cheryl Ferguson, Gary Bartlett, Michael Tamm, Kendall Beaman, Eric Nagler, Max Hailperin, Joe Gottman, Richard Weeks, Valentin Bonnard, Jun He, Tim King, Don Maier, Ted Hill, Mark Harrison, Michael Rubenstein, Mark Rodgers, David Goh, Brenton Cooper, Andy Thomas-Cramer, Antoine Trux, John Wait, Brian Sharon, Liam

Fitzpatrick, Bernd Mohr, Gary Yee, John O'Hanley, Brady Patterson, Christopher Peterson, Feliks Kluzniak, Isi Dunietz, Christopher Creutzi, Ian Cooper, Carl Harris, Mark Stickel, Clay Budin, Panayotis Matsinopoulos, David Smallberg, Herb Sutter, Pajo Misljencevic, Giulio Agostini, Fredrik Blomqvist, Jimmy Snyder, Byrial Jensen, Witold Kuzminski, Kazunobu Kuriyama, Michael Christensen, Jorge Yáñez Teruel, Mark Davis, Marty Rabinowitz, Ares Lagae, and Alexander Medvedev.

An early partial draft of this edition was reviewed by Brian Kernighan, Angelika Langer, Jesse Laeuchli, Roger E. Pedersen, Chris Van Wyk, Nicholas Stroustrup, and Hendrik Schober. Reviewers for a full draft were Leor Zolman, Mike Tsao, Eric Nagler, Gene Gutnik, David Abrahams, Gerhard Kreuzer, Drosos Kourounis, Brian Kernighan, Andrew Kirmse, Balog Pal, Emily Jagdhar, Eugene Kalenkovich, Mike Roze, Enrico Carrara, Benjamin Berck, Jack Reeves, Steve Schirripa, Martin Fallenstedt, Timothy Knox, Yun Bai, Michael Lanzetta, Philipp Janert, Guido Bartolucci, Michael Topic, Jeff Scherpelz, Chris Nauroth, Nishant Mittal, Jeff Somers, Hal Moroff, Vincent Manis, Brandon Chang, Greg Li, Jim Meehan, Alan Geller, Siddhartha Singh, Sam Lee, Sasan Dashtinezhad, Alex Marin, Steve Cai, Thomas Fruchterman, Cory Hicks, David Smallberg, Gunavardhan Kakulapati, Danny Rabbani, Jake Cohen, Hendrik Schober, Paco Viciana, Glenn Kennedy, Jeffrey D. Oldham, Nicholas Stroustrup, Matthew Wilson, Andrei Alexandrescu, Tim Johnson, Leon Matthews, Peter Dulimov, and Kevlin Henney. Drafts of some individual Items were reviewed by Herb Sutter and Attila F. Feher.

Reviewing an unpolished (possibly incomplete) manuscript is demanding work, and doing it under time pressure only makes it harder. I continue to be grateful that so many people have been willing to undertake it for me.

Reviewing is harder still if you have no background in the material being discussed and are expected to catch *every* problem in the manuscript. Astonishingly, some people still choose to be copy editors. Chrysta Meadowbrooke was the copy editor for this book, and her very thorough work exposed many problems that eluded everyone else.

Leor Zolman checked all the code examples against multiple compilers in preparation for the full review, then did it again after I revised the manuscript. If any errors remain, I'm responsible for them, not Leor.

Karl Wiegers and especially Tim Johnson offered rapid, helpful feedback on back cover copy.

Since publication of the first printing, I have incorporated revisions suggested by Jason Ross, Robert Yokota, Bernhard Merkle, Attila Feher, Gerhard Kreuzer, Marcin Sochacki, J. Daniel Smith, Idan Lupinsky, G. Wade Johnson, Clovis Tondo, Joshua Lehrer, T. David Hudson, Phillip Hellewell, Thomas Schell, Eldar Ronen, Ken Kobayashi, Cameron Mac Minn, John Hershberger, Alex Dumov, Vincent Stojanov, Andrew Henrick, Jiongxiong Chen, Balbir Singh, Fraser Ross, Niels Dekker, Harsh Gaurav Vangani, Vasily Poshehonov, Yukitoshi Fujimura, Alex Howlett, Ed Ji Xihuang. Mike Rizzi, Balog Pal, David Solomon, Tony Oliver, Martin Rottinger, Miaohua, and Brian Johnson.

John Wait, my editor for the first two editions of this book, foolishly signed up for another tour of duty in that capacity. His assistant, Denise Mickelsen, adroitly handled my frequent pestering with a pleasant smile. (At least I think she's been smiling. I've never actually seen her.) Julie Nahil drew the short straw and hence became my production manager. She handled the overnight loss of six weeks in the production schedule with remarkable equanimity. John Fuller (her boss) and Marty Rabinowitz (his boss) helped out with production issues, too. Vanessa Moore's official job was to help with FrameMaker issues and PDF preparation, but she also added the entries to Appendix B and formatted it for printing on the inside cover. Solveig Haugland helped with index formatting. Sandra Schroeder and Chuti Prasertsith were responsible for cover design, though Chuti seems to have been the one who had to rework the cover each time I said, “But what about *this* photo with a stripe of *that* color...?” Chanda Leary-Coutu got tapped for the heavy lifting in marketing.

During the months I worked on the manuscript, the TV series *Buffy the Vampire Slayer* often helped me “de-stress” at the end of the day. Only with great restraint have I kept Buffyspeak out of the book.

Kathy Reed taught me programming in 1971, and I'm gratified that we remain friends to this day. Donald French hired me and Moises Lejter to create C++ training materials in 1989 (an act that led to my *really* knowing C++), and in 1991 he engaged me to present them at Stratus Computer. The students in that class encouraged me to write what ultimately became the first edition of this book. Don also introduced me to John Wait, who agreed to publish it.

My wife, Nancy L. Urbano, continues to encourage my writing, even after seven book projects, a CD adaptation, and a dissertation. She has unbelievable forbearance. I couldn't do what I do without her.

From start to finish, our dog, Persephone, has been a companion without equal. Sadly, for much of this project, her companionship has taken the form of an urn in the office. We really miss her.

Introduction

Learning the fundamentals of a programming language is one thing; learning how to design and implement *effective* programs in that language is something else entirely. This is especially true of C++, a language boasting an uncommon range of power and expressiveness. Properly used, C++ can be a joy to work with. An enormous variety of designs can be directly expressed and efficiently implemented. A judiciously chosen and carefully crafted set of classes, functions, and templates can make application programming easy, intuitive, efficient, and nearly error-free. It isn't unduly difficult to write effective C++ programs, *if* you know how to do it. Used without discipline, however, C++ can lead to code that is incomprehensible, unmaintainable, inextensible, inefficient, and just plain wrong.

The purpose of this book is to show you how to use C++ *effectively*. I assume you already know C++ as a *language* and that you have some experience in its use. What I provide here is a guide to using the language so that your software is comprehensible, maintainable, portable, extensible, efficient, and likely to behave as you expect.

The advice I proffer falls into two broad categories: general design strategies, and the nuts and bolts of specific language features. The design discussions concentrate on how to choose between different approaches to accomplishing something in C++. How do you choose between inheritance and templates? Between public and private inheritance? Between private inheritance and composition? Between member and non-member functions? Between pass-by-value and pass-by-reference? It's important to make these decisions correctly at the outset, because a poor choice may not become apparent until much later in the development process, at which point rectifying it is often difficult, time-consuming, and expensive.

Even when you know exactly what you want to do, getting things just right can be tricky. What's the proper return type for assignment operators? When should a destructor be virtual? How should `operator new` behave when it can't find enough memory? It's crucial to sweat details like these, because failure to do so almost always leads to unexpected, possibly mystifying program behavior. This book will help you avoid that.

This is not a comprehensive reference for C++. Rather, it's a collection of 55 specific suggestions (I call them *Items*) for how you can improve your programs and designs. Each Item stands more or less on its own, but most also contain references to other Items. One way to read the book, then, is to start with an Item of interest, then follow its references to see where they lead you.

The book isn't an introduction to C++, either. In Chapter 2, for example, I'm eager to tell you all about the proper implementations of constructors, destructors, and assignment operators, but I assume you already know or can go elsewhere to find out what these functions do and how they are declared. A number of C++ books contain information such as that.

The purpose of *this* book is to highlight those aspects of C++ programming that are often overlooked. Other books describe the different parts of the language. This book tells you how to combine those parts so you end up with effective programs. Other books tell you how to get your programs to compile. This book tells you how to avoid problems that compilers won't tell you about.

At the same time, this book limits itself to *standard* C++. Only features in the official language standard have been used here. Portability is a key concern in this book, so if you're looking for platform-dependent hacks and kludges, this is not the place to find them.

Another thing you won't find in this book is the C++ Gospel, the One True Path to perfect C++ software. Each of the Items in this book provides guidance on how to develop better designs, how to avoid common problems, or how to achieve greater efficiency, but none of the Items is universally applicable. Software design and implementation is a complex task, one colored by the constraints of the hardware, the operating system, and the application, so the best I can do is provide *guidelines* for creating better programs.

If you follow all the guidelines all the time, you are unlikely to fall into the most common traps surrounding C++, but guidelines, by their nature, have exceptions. That's why each Item has an explanation. The explanations are the most important part of the book. Only by understanding the rationale behind an Item can you determine whether it applies to the software you are developing and to the unique constraints under which you toil.

The best use of this book is to gain insight into how C++ behaves, why it behaves that way, and how to use its behavior to your advantage. Blind application of the Items in this book is clearly inappropriate, but at the same time, you probably shouldn't violate any of the guidelines without a good reason.

Terminology

There is a small C++ vocabulary that every programmer should understand. The following terms are important enough that it is worth making sure we agree on what they mean.

A ***declaration*** tells compilers about the name and type of something, but it omits certain details. These are declarations:

Click here to view code image

```
extern int x;                                  // object
declaration

std::size_t numDigits(int number);             // function
declaration

class Widget;                                  // class
declaration

template<typename T>                           // template
declaration
class GraphNode;                               // (see Item
42 for info on
                                               // the use of
"typename")
```

Note that I refer to the integer `x` as an “object,” even though it's of built-in type. Some people reserve the name “object” for variables of user-defined type, but I'm not one of them. Also note that the function `numDigits`' return type is `std::size_t`, i.e., the type `size_t` in namespace `std`. That namespace is where virtually everything in C++'s standard library is located. However, because C's standard library (the one from C89, to be precise) can also be used in C++, symbols inherited from C (such as `size_t`) may exist at global scope, inside `std`, or both, depending on which headers have been `#include`d. In this book, I assume that C++ headers have been `#include`d, and that's why I refer to `std::size_t` instead of just `size_t`. When referring to components of the standard library in prose, I typically omit references to `std`, relying on you to recognize that things like `size_t`, `vector`, and `cout` are in `std`. In example code, I always include `std`, because real code won't compile without it.

`size_t`, by the way, is just a typedef for some unsigned type that C++ uses when counting things (e.g., the number of characters in a `char*`-based string, the number of elements in an STL container, etc.). It's also the type taken by the `operator[]` functions in `vector`, `deque`, and `string`, a convention we'll follow when defining our own `operator[]` functions in Item 3.

Each function's declaration reveals its ***signature***, i.e., its parameter and return types. A function's signature is the same as its type. In the case of `numDigits`, the signature is `std::size_t (int)`, i.e., “function taking an `int` and returning a `std::size_t`.” The official C++ definition of “signature” excludes the function's return type, but in this book, it's more useful to have the return type be considered part of the signature.

A ***definition*** provides compilers with the details a declaration omits. For an object, the definition is where compilers set aside memory for the object. For a function or a function template, the definition provides the code body. For a class or a class template, the definition lists the members of the class or template:

Click here to view code image

```
int x;                                            // object
definition

std::size_t numDigits(int number)                 // function
definition.
{                                                 // (This
function returns
  std::size_t digitsSoFar = 1;                    // the
number of digits
                                                  // in its
parameter.)
  while ((number /= 10) != 0) ++digitsSoFar;

  return digitsSoFar;
}

class Widget {                                    // class
definition
public:
  Widget();
  ~Widget();
  ...
};

template<typename T>                       // template
definition
class GraphNode {
public:
  GraphNode();
  ~GraphNode();
  ...
};
```

Initialization is the process of giving an object its first value. For objects generated from structs and classes, initialization is performed by constructors. A ***default constructor*** is one that can be called without any arguments. Such a constructor either has no parameters or has a default value for every parameter.

Click here to view code image

```
class A {
public:
  A();                                     // default
constructor
};

class B {
public:
  explicit B(int x = 0, bool b = true);    // default
constructor; see below
};                                         // for info on
"explicit"


class C {
public:
  explicit C(int x);                       // not a default
constructor
};
```

The constructors for classes `B` and `C` are declared `explicit` here. That prevents them from being used to perform implicit type conversions, though they may still be used for explicit type conversions:

Click here to view code image

```
void doSomething(B bObject);          // a function taking an
object of
                                      // type B

B bObj1;                              // an object of type B

doSomething(bObj1);                   // fine, passes a B to
doSomething

B bObj2(28);                          // fine, creates a B
from the int 28
                                      // (the bool defaults to

true)

doSomething(28);                      // error! doSomething
takes a B,
                                      // not an int, and there

is no
                                      // implicit conversion

from int to B

doSomething(B(28));                   // fine, uses the B
constructor to
                                      // explicitly convert

(i.e., cast) the
                                      // int to a B for this

call. (See
                                      // Item 27 for info on

casting.)
```

Constructors declared `explicit` are usually preferable to non-`explicit` ones, because they prevent compilers from performing unexpected (often unintended) type conversions. Unless I have a good reason for allowing a constructor to be used for implicit type conversions, I declare it `explicit`. I encourage you to follow the same policy.

Please note how I've highlighted the cast in the example above. Throughout this book, I use such highlighting to call your attention to material that is particularly noteworthy. (I also highlight chapter numbers, but that's just because I think it looks nice.)

The ***copy constructor*** is used to initialize an object with a different object of the same type, and the ***copy assignment operator*** is used to copy the value from one object to another of the same type:

Click here to view code image

```
class Widget {
public:
  Widget();                             // default
constructor
  Widget(const Widget& rhs);            // copy
constructor
  Widget& operator=(const Widget& rhs); // copy
assignment operator
  ...
};
Widget w1;                              // invoke default
constructor

Widget w2(w1);                          // invoke copy
constructor

w1 = w2;                                // invoke copy
                                        // assignment
operator
```

Read carefully when you see what appears to be an assignment, because the "`=`" syntax can also be used to call the copy constructor:

```
Widget w3 = w2;                       // invoke copy
constructor!
```

Fortunately, copy construction is easy to distinguish from copy assignment. If a new object is being defined (such as `w3` in the statement above), a constructor has to be called; it can't be an assignment. If no new object is being defined (such as in the "`w1 = w2`" statement above), no constructor can be involved, so it's an assignment.

The copy constructor is a particularly important function, because it defines how an object is passed by value. For example, consider this:

Click here to view code image

```
bool hasAcceptableQuality(Widget w);

...
Widget aWidget;
if (hasAcceptableQuality(aWidget)) ...
```

The parameter `w` is passed to `hasAcceptableQuality` by value, so in the call above, `aWidget` is copied into `w`. The copying is done by `Widget`'s copy constructor. Pass-by-value *means* “call the copy constructor.” (However, it's generally a bad idea to pass user-defined types by value. Pass-by-reference-to-`const` is typically a better choice. For details, see Item 20.)

The ***STL*** is the Standard Template Library, the part of C++'s standard library devoted to containers (e.g., `vector`, `list`, `set`, `map`, etc.), iterators (e.g., `vector<int>::iterator`, `set<string>::iterator`, etc.), algorithms (e.g., `for_each`, `find`, `sort`, etc.), and related functionality. Much of that related functionality has to do with ***function objects***: objects that act like functions. Such objects come from classes that overload `operator()`, the function call operator. If you're unfamiliar with the STL, you'll want to have a decent reference available as you read this book, because the STL is too useful for me not to take advantage of it. Once you've used it a little, you'll feel the same way.

Programmers coming to C++ from languages like Java or C# may be surprised at the notion of ***undefined behavior.*** For a variety of reasons, the behavior of some constructs in C++ is literally not defined: you can't reliably predict what will happen at runtime. Here are two examples of code with undefined behavior:

Click here to view code image

```
int *p = 0;                            // p is a null pointer

std::cout << *p;                       // dereferencing a
null pointer
                                       // yields undefined

behavior

char name[] = "Darla";                 // name is an array of
size 6 (don't
                                       // forget the trailing

null!)
```

```
char c = name[10];                  // referring to an
invalid array index
                                    // yields undefined
behavior
```

To emphasize that the results of undefined behavior are not predictable and may be very unpleasant, experienced C++ programmers often say that programs with undefined behavior can erase your hard drive. It's true: a program with undefined behavior *could* erase your hard drive. But it's not probable. More likely is that the program will behave erratically, sometimes running normally, other times crashing, still other times producing incorrect results. Effective C++ programmers do their best to steer clear of undefined behavior. In this book, I point out a number of places where you need to be on the lookout for it.

Another term that may confuse programmers coming to C++ from another language is ***interface***. Java and the .NET languages offer Interfaces as a language element, but there is no such thing in C++, though Item 31 discusses how to approximate them. When I use the term “interface,” I'm generally talking about a function's signature, about the accessible elements of a class (e.g., a class's “public interface,” “protected interface,” or “private interface”), or about the expressions that must be valid for a template's type parameter (see Item 41). That is, I'm talking about interfaces as a fairly general design idea.

A ***client*** is someone or something that uses the code (typically the interfaces) you write. A function's clients, for example, are its users: the parts of the code that call the function (or take its address) as well as the humans who write and maintain such code. The clients of a class or a template are the parts of the software that use the class or template, as well as the programmers who write and maintain that code. When discussing clients, I typically focus on programmers, because programmers can be confused, misled, or annoyed by bad interfaces. The code they write can't be.

You may not be used to thinking about clients, but I'll spend a good deal of time trying to convince you to make their lives as easy as you can. After all, you are a client of the software other people develop. Wouldn't you want those people to make things easy for you? Besides, at some point you'll almost certainly find yourself in the position of being your own client (i.e., using code you wrote), and at that point, you'll be glad you kept client concerns in mind when developing your interfaces.

In this book, I often gloss over the distinction between functions and function templates and between classes and class templates. That's because what's true about one is often true about the other. In situations where this is not the case, I distinguish among classes, functions, and the templates that give rise to classes and functions.

When referring to constructors and destructors in code comments, I sometimes use the abbreviations ***ctor*** and ***dtor***.

Naming Conventions

I have tried to select meaningful names for objects, classes, functions, templates, etc., but the meanings behind some of my names may not be immediately apparent. Two of my favorite parameter names, for example, are `lhs` and `rhs`. They stand for “left-hand side” and “right-hand side,” respectively. I often use them as parameter names for functions implementing binary operators, e.g., `operator==` and `operator*`. For example, if `a` and `b` are objects representing rational numbers, and if `Rational` objects can be multiplied via a non-member `operator*` function (as Item 24 explains is likely to be the case), the expression

```
a * b
```

is equivalent to the function call

```
operator*(a,b)
```

In Item 24, I declare `operator*` like this:

```
const Rational operator*(const Rational& lhs, const Rational&
rhs);
```

As you can see, the left-hand operand, `a`, is known as `lhs` inside the function, and the right-hand operand, `b`, is known as `rhs`.

For member functions, the left-hand argument is represented by the `this` pointer, so sometimes I use the parameter name `rhs` by itself. You may have noticed this in the declarations for some `Widget` member functions on page 5. Which reminds me. I often use the `Widget` class in examples. “Widget” doesn't mean anything. It's just a name I sometimes use when I need an example class name. It has nothing to do with widgets in GUI toolkits.

I often name pointers following the rule that a pointer to an object of type `T` is called `pt`, “pointer to T.” Here are some examples:

[Click here to view code image]

```
Widget *pw;                          // pw = ptr to Widget

class Airplane;
Airplane *pa;                        // pa = ptr to Airplane

class GameCharacter;
GameCharacter *pgc;                  // pgc = ptr to
GameCharacter
```

I use a similar convention for references: `rw` might be a reference to a `Widget` and `ra` a reference to an `Airplane`.

I occasionally use the name `mf` when I'm talking about member functions.

Threading Considerations

As a language, C++ has no notion of threads — no notion of concurrency of any kind, in fact. Ditto for C++'s standard library. As far as C++ is concerned, multithreaded programs don't exist.

And yet they do. My focus in this book is on standard, portable C++, but I can't ignore the fact that thread safety is an issue many programmers confront. My approach to dealing with this chasm between standard C++ and reality is to point out places where the C++ constructs I examine are likely to cause problems in a threaded environment. That doesn't make this a book on multithreaded programming with C++. Far from it. Rather, it makes it a book on C++ programming that, while largely limiting itself to single-threaded considerations, acknowledges the existence of multithreading and tries to point out places where thread-aware programmers need to take particular care in evaluating the advice I offer.

If you're unfamiliar with multithreading or have no need to worry about it, you can ignore my threading-related remarks. If you are programming a threaded application or library, however, remember that my comments are little more than a starting point for the issues you'll need to address when using C++.

TR1 and Boost

You'll find references to TR1 and Boost throughout this book. Each has an Item that describes it in some detail (Item 54 for TR1, Item 55 for Boost), but, unfortunately, these Items are at the end of the book. (They're there because it works better that way. Really. I tried them in a number of other places.) If you like, you can turn to those Items and read them now, but if you'd prefer to start the book at the beginning instead of the end, the following executive summary will tide you over:

- TR1 (“Technical Report 1”) is a specification for new functionality being added to C++'s standard library. This functionality takes the form of new class and function templates for things like hash tables, reference-counting smart pointers, regular expressions, and more. All TR1 components are in the namespace `tr1` that's nested inside the namespace `std`.
- Boost is an organization and a web site (http://boost.org) offering portable, peer-reviewed, open source C++ libraries. Most TR1 functionality is based on work done at Boost, and until compiler vendors include TR1 in their C++ library distributions, the Boost web site is likely to remain the first stop for developers looking for TR1 implementations. Boost offers more than is available in TR1, however, so it's worth knowing about in any case.

1. Accustoming Yourself to C++

Regardless of your programming background, C++ is likely to take a little getting used to. It's a powerful language with an enormous range of features, but before you can harness that power and make effective use of those features, you have to accustom yourself to C++'s way of doing things. This entire book is about how to do that, but some things are more fundamental than others, and this chapter is about some of the most fundamental things of all.

Item 1: View C++ as a federation of languages

In the beginning, C++ was just C with some object-oriented features tacked on. Even C++'s original name, “C with Classes,” reflected this simple heritage.

As the language matured, it grew bolder and more adventurous, adopting ideas, features, and programming strategies different from those of C with Classes. Exceptions required different approaches to structuring functions (see Item 29). Templates gave rise to new ways of thinking about design (see Item 41), and the STL defined an approach to extensibility unlike any most people had ever seen.

Today's C++ is a *multiparadigm programming language*, one supporting a combination of procedural, object-oriented, functional, generic, and metaprogramming features. This power and flexibility make C++ a tool without equal, but can also cause some confusion. All the “proper usage” rules seem to have exceptions. How are we to make sense of such a language?

The easiest way is to view C++ not as a single language but as a federation of related languages. Within a particular sublanguage, the rules tend to be simple, straightforward, and easy to remember. When you move from one sublanguage to another, however, the rules may change. To make sense of C++, you have to recognize its primary sublanguages. Fortunately, there are only four:

- **C.** Way down deep, C++ is still based on C. Blocks, statements, the preprocessor, built-in data types, arrays, pointers, etc., all come from C.

In many cases, C++ offers approaches to problems that are superior to their C counterparts (e.g., see Items 2 (alternatives to the preprocessor) and 13 (using objects to manage resources)), but when you find yourself working with the C part of C++, the rules for effective programming reflect C's more limited scope: no templates, no exceptions, no overloading, etc.

- **Object-Oriented C++.** This part of C++ is what C with Classes was all about: classes (including constructors and destructors), encapsulation, inheritance, polymorphism, virtual functions (dynamic binding), etc. This is the part of C++ to which the classic rules for object-oriented design most directly apply.
- **Template C++.** This is the generic programming part of C++, the one that most programmers have the least experience with. Template considerations pervade C++, and it's not uncommon for rules of good programming to include special template-only clauses (e.g., see Item 46 on facilitating type conversions in calls to template functions). In fact, templates are so powerful, they give rise to a completely new programming paradigm, *template metaprogramming* (TMP). Item 48 provides an overview of TMP, but unless you're a hard-core template junkie, you need not worry about it. The rules for TMP rarely interact with mainstream C++ programming.
- **The STL.** The STL is a template library, of course, but it's a very special template library. Its conventions regarding containers, iterators, algorithms, and function objects mesh beautifully, but templates and libraries can be built around other ideas, too. The STL has particular ways of doing things, and when you're working with the STL, you need to be sure to follow its conventions.

Keep these four sublanguages in mind, and don't be surprised when you encounter situations where effective programming requires that you change strategy when you switch from one sublanguage to another. For example, pass-by-value is generally more efficient than pass-by-reference for built-in (i.e., C-like) types, but when you move from the C part of C++ to Object-Oriented C++, the existence of user-defined constructors and destructors means that pass-by-reference-to-`const` is usually better. This is especially the case when working in Template C++, because there, you don't even know the type of object you're dealing with. When you cross into the STL, however, you know that iterators and function objects are modeled on pointers in C, so for iterators and function objects in the STL, the old C pass-by-value rule applies again. (For all the details on choosing among parameter-passing options, see Item 20.)

C++, then, isn't a unified language with a single set of rules; it's a federation of four sublanguages, each with its own conventions. Keep these sublanguages in mind, and you'll find that C++ is a lot easier to understand.

Things to Remember

- Rules for effective C++ programming vary, depending on the part of C++ you are using.

Item 2: Prefer `const`s, `enum`s, and `inline`s to `#define`s

This Item might better be called “prefer the compiler to the preprocessor,” because `#define` may be treated as if it's not part of the language *per se*. That's one of its problems. When you do something like this,

```
#define ASPECT_RATIO 1.653
```

the symbolic name `ASPECT_RATIO` may never be seen by compilers; it may be removed by the preprocessor before the source code ever gets to a compiler. As a result, the name `ASPECT_RATIO` may not get entered into the symbol table. This can be confusing if you get an error during compilation involving the use of the constant, because the error message may refer to `1.653`, not `ASPECT_RATIO`. If `ASPECT_RATIO` were defined in a header file you didn't write, you'd have no idea where that `1.653` came from, and you'd waste time tracking it down. This problem can also crop up in a symbolic debugger, because, again, the name you're programming with may not be in the symbol table.

The solution is to replace the macro with a constant:

```
const double AspectRatio = 1.653;   // uppercase names are
usually for
                                    // macros, hence the name
change
```

As a language constant, `AspectRatio` is definitely seen by compilers and is certainly entered into their symbol tables. In addition, in the case of a floating point constant (such as in this example), use of the constant may yield smaller code than using a `#define`. That's because the preprocessor's blind substitution of the macro name `ASPECT_RATIO` with 1.653 could result in multiple copies of 1.653 in your object code, while the use of the constant `AspectRatio` should never result in more than one copy.

When replacing `#define`s with constants, two special cases are worth mentioning. The first is defining constant pointers. Because constant definitions are typically put in header files (where many different source files will include them), it's important that the *pointer* be declared `const`, usually in addition to what the pointer points to. To define a constant `char*`-based string in a header file, for example, you have to write `const` *twice*:

```
const char * const authorName = "Scott Meyers";
```

For a complete discussion of the meanings and uses of `const`, especially in conjunction with pointers, see Item 3. However, it's worth reminding you here that `string` objects are generally preferable to their `char*`-based progenitors, so `authorName` is often better defined this way:

```
const std::string authorName("Scott Meyers");
```

The second special case concerns class-specific constants. To limit the scope of a constant to a class, you must make it a member, and to ensure there's at most one copy of the constant, you must make it a *static* member:

Click here to view code image

```
class GamePlayer {
private:
  static const int NumTurns = 5;      // constant declaration
  int scores[NumTurns];               // use of constant
  ...
};
```

What you see above is a *declaration* for `NumTurns`, not a definition. Usually, C++ requires that you provide a definition for anything you use, but class-specific constants that are static and of integral type (e.g., integers, `char`s, `bool`s) are an exception. As long as you don't take their address, you can declare them and use them without providing a definition. If you do take the address of a class constant, or if your compiler incorrectly insists on a definition even if you don't take the address, you provide a separate definition like this:

Click here to view code image

```
const int GamePlayer::NumTurns;     // definition of
NumTurns; see
                                    // below for why no value
is given
```

You put this in an implementation file, not a header file. Because the initial value of class constants is provided where the constant is declared (e.g., `NumTurns` is initialized to 5 when it is declared), no initial value is permitted at the point of definition.

Note, by the way, that there's no way to create a class-specific constant using a `#define`, because `#define`s don't respect scope. Once a macro is defined, it's in force for the rest of the compilation (unless it's `#undef`ed somewhere along the line). Which means that not only can't `#define`s be used for class-specific constants, they also can't be used to provide any kind of encapsulation, i.e., there is no such thing as a “private” `#define`. Of course, `const` data members can be encapsulated; `NumTurns` is.

Older compilers may not accept the syntax above, because it used to be illegal to provide an initial value for a static class member at its point of declaration. Furthermore, in-class initialization is allowed only for integral types and only for constants. In cases where the above syntax can't be used, you put the initial value at the point of definition:

Click here to view code image

```
class CostEstimate {
private:

  static const double FudgeFactor;       // declaration of
static class
  ...                                    // constant; goes in
header file
};
const double                             // definition of
static class
  CostEstimate::FudgeFactor = 1.35;      // constant; goes in
impl. file
```

This is all you need almost all the time. The only exception is when you need the value of a class constant during compilation of the class, such as in the declaration of the array `GamePlayer::scores` above (where compilers insist on knowing the size of the array during compilation). Then the accepted way to compensate for compilers that (incorrectly) forbid the in-class specification of initial values for static integral class constants is to use what is affectionately (and non-pejoratively) known as "the enum hack." This technique takes advantage of the fact that the values of an enumerated type can be used where `int`s are expected, so `GamePlayer` could just as well be defined like this:

Click here to view code image

```
class GamePlayer {
private:
  enum { NumTurns = 5 };          // "the enum hack" — makes
                                  // NumTurns a symbolic name
for 5

  int scores[NumTurns];           // fine

  ...

};
```

The enum hack is worth knowing about for several reasons. First, the enum hack behaves in some ways more like a `#define` than a `const` does, and sometimes that's what you want. For example, it's legal to take the address of a `const`, but it's not legal to take the address of an enum, and it's typically not legal to take the address of a `#define`, either. If you don't want to let people get a pointer or reference to one of your integral constants, an enum is a good way to enforce that constraint. (For more on enforcing design constraints through coding decisions, consult Item 18.) Also, though good compilers won't set aside storage for `const` objects of integral types (unless you create a pointer or reference to the object), sloppy compilers may, and you may not be willing to set aside memory for such objects. Like `#define`s, enums never result in that kind of unnecessary memory allocation.

A second reason to know about the enum hack is purely pragmatic. Lots of code employs it, so you need to recognize it when you see it. In fact, the enum hack is a fundamental technique of template metaprogramming (see

Item 48).

Getting back to the preprocessor, another common (mis)use of the `#define` directive is using it to implement macros that look like functions but that don't incur the overhead of a function call. Here's a macro that calls some function `f` with the greater of the macro's arguments:

```
// call f with the maximum of a and b
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))
```

Macros like this have so many drawbacks, just thinking about them is painful.

Whenever you write this kind of macro, you have to remember to parenthesize all the arguments in the macro body. Otherwise you can run into trouble when somebody calls the macro with an expression. But even if you get that right, look at the weird things that can happen:

Click here to view code image

```
int a = 5, b = 0;

CALL_WITH_MAX(++a, b);          // a is incremented twice
CALL_WITH_MAX(++a, b+10);       // a is incremented once
```

Here, the number of times that `a` is incremented before calling `f` depends on what it is being compared with!

Fortunately, you don't need to put up with this nonsense. You can get all the efficiency of a macro plus all the predictable behavior and type safety of a regular function by using a template for an inline function (see Item 30):

Click here to view code image

```
template<typename T>                               // because
we don't
inline void callWithMax(const T& a, const T& b)    // know
what T is, we
{                                                  // pass by
reference-to-
  f(a > b ? a : b);                                // const —
see Item 20
}
```

This template generates a whole family of functions, each of which takes two objects of the same type and calls `f` with the greater of the two objects. There's no need to parenthesize parameters inside the function body, no need to worry about evaluating parameters multiple times, etc. Furthermore, because `callWithMax` is a real function, it obeys scope and access rules. For example, it makes perfect sense to talk about an inline function that is private to a class. In general, there's just no way to do that with a macro.

Given the availability of `const`s, `enum`s, and `inline`s, your need for the preprocessor (especially `#define`) is reduced, but it's not eliminated. `#include` remains essential, and `#ifdef`/`#ifndef` continue to play important roles in controlling compilation. It's not yet time to retire the preprocessor, but you should definitely give it long and frequent vacations.

Things to Remember

- For simple constants, prefer `const` objects or enums to `#define`s.
- For function-like macros, prefer inline functions to `#define`s.

Item 3: Use `const` whenever possible

The wonderful thing about `const` is that it allows you to specify a semantic constraint — a particular object should *not* be modified — and compilers will enforce that constraint. It allows you to communicate to both compilers and other programmers that a value should remain invariant. Whenever that is true, you should be sure to say so, because that way you enlist your compilers' aid in making sure the constraint isn't violated.

The `const` keyword is remarkably versatile. Outside of classes, you can use it for constants at global or namespace scope (see Item 2), as well as for objects declared `static` at file, function, or block scope. Inside classes, you can use it for both static and non-static data members. For pointers, you can specify whether the pointer itself is `const`, the data it points to is `const`, both, or neither:

Click here to view code image

```
char greeting[] = "Hello";

char *p = greeting;                    // non-const pointer,
                                       // non-const data
```

```
const char *p = greeting;               // non-const pointer,
                                        // const data

char * const p = greeting;              // const pointer,
                                        // non-const data

const char * const p = greeting;        // const pointer,
                                        // const data
```

This syntax isn't as capricious as it may seem. If the word `const` appears to the left of the asterisk, what's *pointed to* is constant; if the word `const` appears to the right of the asterisk, the *pointer itself* is constant; if `const` appears on both sides, both are constant.[‡]

When what's pointed to is constant, some programmers list `const` before the type. Others list it after the type but before the asterisk. There is no difference in meaning, so the following functions take the same parameter type:

Click here to view code image

```
void f1(const Widget *pw);          // f1 takes a pointer to a
                                    // constant Widget object

void f2(Widget const *pw);          // so does f2
```

Because both forms exist in real code, you should accustom yourself to both of them.

STL iterators are modeled on pointers, so an `iterator` acts much like a `T*` pointer. Declaring an `iterator const` is like declaring a pointer `const` (i.e., declaring a `T* const` pointer): the `iterator` isn't allowed to point to something different, but the thing it points to may be modified. If you want an iterator that points to something that can't be modified (i.e., the STL analogue of a `const T*` pointer), you want a `const_iterator`:

Click here to view code image

```
std::vector<int> vec;
...

const std::vector<int>::iterator iter =     // iter acts like
a T* const
  vec.begin();
```

```
*iter = 10;                                     // OK, changes
what iter points to
++iter;                                        // error! iter is
const

std::vector<int>::const_iterator cIter =   //cIter acts like
a const T*
  vec.begin();
*cIter = 10;                                   // error! *cIter
is const
++cIter;                                      // fine, changes
cIter
```

Some of the most powerful uses of `const` stem from its application to function declarations. Within a function declaration, `const` can refer to the function's return value, to individual parameters, and, for member functions, to the function as a whole.

Having a function return a constant value often makes it possible to reduce the incidence of client errors without giving up safety or efficiency. For example, consider the declaration of the `operator*` function for rational numbers that is explored in Item 24:

Click here to view code image

```
class Rational { ... };

const Rational operator*(const Rational& lhs, const Rational&
rhs);
```

Many programmers squint when they first see this. Why should the result of `operator*` be a `const` object? Because if it weren't, clients would be able to commit atrocities like this:

Click here to view code image

```
Rational a, b, c;

...

(a * b) = c;                             // invoke operator= on
the
                                         // result of a*b!
```

I don't know why any programmer would want to make an assignment to the product of two numbers, but I do know that many programmers have tried to do it without wanting to. All it takes is a simple typo (and a type that can be implicitly converted to `bool`):

```
if (a * b = c) ...                       // oops, meant to do
a comparison!
```

Such code would be flat-out illegal if `a` and `b` were of a built-in type. One of the hallmarks of good user-defined types is that they avoid gratuitous incompatibilities with the built-ins (see also Item 18), and allowing assignments to the product of two numbers seems pretty gratuitous to me. Declaring `operator*`'s return value `const` prevents it, and that's why it's The Right Thing To Do.

There's nothing particularly new about `const` parameters — they act just like local `const` objects, and you should use both whenever you can. Unless you need to be able to modify a parameter or local object, be sure to declare it `const`. It costs you only the effort to type six characters, and it can save you from annoying errors such as the "I meant to type '==' but I accidently typed '='" mistake we just saw.

`const` Member Functions

The purpose of `const` on member functions is to identify which member functions may be invoked on `const` objects. Such member functions are important for two reasons. First, they make the interface of a class easier to understand. It's important to know which functions may modify an object and which may not. Second, they make it possible to work with `const` objects. That's a critical aspect of writing efficient code, because, as Item 20 explains, one of the fundamental ways to improve a C++ program's performance is to pass objects by reference-to-`const`. That technique is viable only if there are `const` member functions with which to manipulate the resulting `const`-qualified objects.

Many people overlook the fact that member functions differing *only* in their constness can be overloaded, but this is an important feature of C++. Consider a class for representing a block of text:

Click here to view code image

```
class TextBlock {
public:
  ...

  const char& operator[](std::size_t position) const   //
operator[] for
  { return text[position]; }                           //
const objects

  char& operator[](std::size_t position)               //
operator[] for
  { return text[position]; }                           //
non-const objects

private:
   std::string text;
};
```

`TextBlock`'s `operator[]`s can be used like this:

Click here to view code image

```
TextBlock tb("Hello");
std::cout << tb[0];                      // calls non-const
                                               //
TextBlock::operator[]
const TextBlock ctb("World");
std::cout << ctb[0];                     // calls const
TextBlock::operator[]
```

Incidentally, `const` objects most often arise in real programs as a result of being passed by pointer- or reference-to-`const`. The example of `ctb` above is artificial. This is more realistic:

Click here to view code image

```
void print(const TextBlock& ctb)        // in this function,
ctb is const
{
  std::cout << ctb[0];                  // calls const
TextBlock::operator[]
  ...
}
```

By overloading `operator[]` and giving the different versions different return types, you can have `const` and non-`const` `TextBlock`s handled differently:

Click here to view code image

```
std::cout << tb[0];                   // fine — reading a
                                      // non-const TextBlock


tb[0] = 'x';                           // fine — writing a
                                      // non-const TextBlock

std::cout << ctb[0];                  // fine — reading a
                                      // const TextBlock

ctb[0] = 'x';                          // error! — writing a
                                      // const TextBlock
```

Note that the error here has only to do with the *return type* of the `operator[]` that is called; the calls to `operator[]` themselves are all fine. The error arises out of an attempt to make an assignment to a `const char&`, because that's the return type from the `const` version of `operator[]`.

Also note that the return type of the non-`const` `operator[]` is a *reference* to a `char` — a `char` itself would not do. If `operator[]` did return a simple `char`, statements like this wouldn't compile:

```
tb[0] = 'x';
```

That's because it's never legal to modify the return value of a function that returns a built-in type. Even if it were legal, the fact that C++ returns objects by value (see Item 20) would mean that a *copy* of `tb.text[0]` would be modified, not `tb.text[0]` itself, and that's not the behavior you want.

Let's take a brief time-out for philosophy. What does it mean for a member function to be `const`? There are two prevailing notions: *bitwise constness* (also known as *physical constness*) and *logical constness*.

The bitwise `const` camp believes that a member function is `const` if and only if it doesn't modify any of the object's data members (excluding those that are static), i.e., if it doesn't modify any of the bits inside the object. The nice thing about bitwise constness is that it's easy to detect violations: compilers just look for assignments to data members. In fact, bitwise constness is C++'s definition of constness, and a `const` member function isn't allowed to modify any of the non-static data members of the object on which it is invoked.

Unfortunately, many member functions that don't act very `const` pass the bitwise test. In particular, a member function that modifies what a pointer *points to* frequently doesn't act `const`. But if only the *pointer* is in the object, the function is bitwise `const`, and compilers won't complain. That can lead to counterintuitive behavior. For example, suppose we have a `TextBlock`-like class that stores its data as a `char*` instead of a `string`, because it needs to communicate through a C API that doesn't understand `string` objects.

Click here to view code image

```
class CTextBlock {
public:
  ...

  char& operator[](std::size_t position) const    //
inappropriate (but bitwise
  { return pText[position]; }                     // const)
declaration of
                                                  //
operator[]
private:
  char *pText;
};
```

This class (inappropriately) declares `operator[]` as a `const` member function, even though that function returns a reference to the object's internal data (a topic treated in depth in Item 28). Set that aside and note that `operator[]`'s implementation doesn't modify `pText` in any way. As a result, compilers will happily generate code for `operator[]`; it is, after all, bitwise `const`, and that's all compilers check for. But look what it allows to happen:

Click here to view code image

```
const CTextBlock cctb("Hello");         // declare constant
object

char *pc = &cctb[0];                    // call the const
operator[] to get a
                                        // pointer to cctb's
data
```

```
*pc = 'J';                              // cctb now has the
value "Jello"
```

Surely there is something wrong when you create a constant object with a particular value and you invoke only `const` member functions on it, yet you still change its value!

This leads to the notion of logical constness. Adherents to this philosophy — and you should be among them —a rgue that a `const` member function might modify some of the bits in the object on which it's invoked, but only in ways that clients cannot detect. For example, your `CTextBlock` class might want to cache the length of the textblock whenever it's requested:

Click here to view code image

```
class CTextBlock {
public:

  ...

  std::size_t length() const;

private:
  char *pText;
  std::size_t textLength;            // last calculated
length of textblock
  bool lengthIsValid;                // whether length is
currently valid
};

std::size_t CTextBlock::length() const
{
  if (!lengthIsValid) {
    textLength = std::strlen(pText);  // error! can't assign
to textLength
    lengthIsValid = true;             // and lengthIsValid in
a const
  }                                   // member function

  return textLength;
}
```

This implementation of `length` is certainly not bitwise `const` — both `textLength` and `lengthIsValid` may be modified — yet it seems as though it should be valid for `const CTextBlock` objects. Compilers disagree. They insist on bitwise constness. What to do?

The solution is simple: take advantage of C++'s `const`-related wiggle room known as `mutable`. `mutable` frees non-static data members from the constraints of bitwise constness:

Click here to view code image

```
class CTextBlock {
public:

  ...

  std::size_t length() const;

private:
  char *pText;

  mutable std::size_t textLength;         // these data
members may
  mutable bool lengthIsValid;             // always be
modified, even in
};                                        // const member
functions

std::size_t CTextBlock::length() const
{
  if (!lengthIsValid) {
    textLength = std::strlen(pText);      // now fine
    lengthIsValid = true;                 // also fine
  }

  return textLength;
}
```

Avoiding Duplication in `const` and Non-`const` Member Functions

`mutable` is a nice solution to the bitwise-constness-is-not-what-I-had-in-mind problem, but it doesn't solve all `const`-related difficulties. For example, suppose that `operator[]` in `TextBlock` (and `CTextBlock`) not only returned a reference to the appropriate character, it also performed bounds checking, logged access information, maybe even did data integrity validation. Putting all this in both the `const` and the non-`const` `operator[]` functions (and not fretting that we now have implicitly inline functions of nontrivial length — see Item 30) yields this kind of monstrosity:

Click here to view code image

```
class TextBlock {
public:

  ...

  const char& operator[](std::size_t position) const
  {
    ...                                 // do bounds checking
    ...                                 // log access data
    ...                                 // verify data
integrity
    return text[position];
  }

  char& operator[](std::size_t position)
  {
    ...                                 // do bounds checking
    ...                                 // log access data
    ...                                 // verify data
integrity
    return text[position];
  }

private:
   std::string text;
};
```

Ouch! Can you say code duplication, along with its attendant compilation time, maintenance, and code-bloat headaches? Sure, it's possible to move all the code for bounds checking, etc. into a separate member function (private, naturally) that both versions of `operator[]` call, but you've still got the duplicated calls to that function and you've still got the duplicated `return` statement code.

What you really want to do is implement `operator[]` functionality once and use it twice. That is, you want to have one version of `operator[]` call the other one. And that brings us to casting away constness.

As a general rule, casting is such a bad idea, I've devoted an entire Item to telling you not to do it (Item 27), but code duplication is no picnic, either. In this case, the `const` version of `operator[]` does exactly what the non-`const` version does, it just has a `const`-qualified return type. Casting away the `const` on the return value is safe, in this case, because whoever called the non-`const` `operator[]` must have had a non-`const` object in the first place. Otherwise they couldn't have called a non-`const` function. So having the non-`const` `operator[]` call the `const` version is a safe way to avoid code duplication, even though it requires a cast. Here's the code, but it may be clearer after you read the explanation that follows:

Click here to view code image

```
class TextBlock {
public:

  ...

  const char& operator[](std::size_t position) const     //
same as before
  {
    ...
    ...
    ...
    return text[position];
  }

  char& operator[](std::size_t position)         // now just
calls const op[]
  {
    return
      const_cast<char&>(                         // cast away
const on
                                                 // op[]'s
return type;
        static_cast<const TextBlock&>(*this)     // add const
to *this's type;
          [position]                            // call const
version of op[]
      );
  }

...

};
```

As you can see, the code has two casts, not one. We want the non-`const` `operator[]` to call the `const` one, but if, inside the non-`const` `operator[]`, we just call `operator[]`, we'll recursively call ourselves. That's only entertaining the first million or so times. To avoid infinite recursion, we have to specify that we want to call the `const operator[]`, but there's no direct way to do that. Instead, we cast `*this` from its native type of `TextBlock&` to `const TextBlock&`. Yes, we use a cast to *add* `const`! So we have two casts: one to add `const` to `*this` (so that our call to `operator[]` will call the `const` version), the second to remove the `const` from the `const operator[]`'s return value.

The cast that adds `const` is just forcing a safe conversion (from a non-`const` object to a `const` one), so we use a `static_cast` for that. The one that removes `const` can be accomplished only via a `const_cast`, so we don't really have a choice there. (Technically, we do. A C-style cast would also work, but, as I explain in Item 27, such casts are rarely the right choice. If you're unfamiliar with `static_cast` or `const_cast`, Item 27 contains an overview.)

On top of everything else, we're calling an operator in this example, so the syntax is a little strange. The result may not win any beauty contests, but it has the desired effect of avoiding code duplication by implementing the non-`const` version of `operator[]` in terms of the `const` version. Whether achieving that goal is worth the ungainly syntax is something only you can determine, but the technique of implementing a non-`const` member function in terms of its `const` twin is definitely worth knowing.

Even more worth knowing is that trying to do things the other way around — avoiding duplication by having the `const` version call the non-`const` version — is *not* something you want to do. Remember, a `const` member function promises never to change the logical state of its object, but a non-`const` member function makes no such promise. If you were to call a non-`const` function from a `const` one, you'd run the risk that the object you'd promised not to modify would be changed. That's why having a `const` member function call a non-`const` one is wrong: the object could be changed. In fact, to get the code to compile, you'd have to use a `const_cast` to get rid of the `const` on `*this`, a clear sign of trouble. The reverse calling sequence — the one we used above — is safe: the non-`const` member function can do whatever it wants with an object, so calling a `const` member function imposes no risk. That's why a `static_cast` works on `*this` in that case: there's no `const`-related danger.

As I noted at the beginning of this Item, `const` is a wonderful thing. On pointers and iterators; on the objects referred to by pointers, iterators, and references; on function parameters and return types; on local variables; and on member functions, `const` is a powerful ally. Use it whenever you can. You'll be glad you did.

Things to Remember

- Declaring something `const` helps compilers detect usage errors. `const` can be applied to objects at any scope, to function parameters and return types, and to member functions as a whole.
- Compilers enforce bitwise constness, but you should program using logical constness.
- When `const` and non-`const` member functions have essentially identical implementations, code duplication can be avoided by having the non-`const` version call the `const` version.

Item 4: Make sure that objects are initialized before they're used

C++ can seem rather fickle about initializing the values of objects. For example, if you say this,

```
int x;
```

in some contexts, `x` is guaranteed to be initialized (to zero), but in others, it's not. If you say this,

```
class Point {
  int x, y;
};

...
```

```
Point p;
```

`p`'s data members are sometimes guaranteed to be initialized (to zero), but sometimes they're not. If you're coming from a language where uninitialized objects can't exist, pay attention, because this is important.

Reading uninitialized values yields undefined behavior. On some platforms, the mere act of reading an uninitialized value can halt your program. More typically, the result of the read will be semi-random bits, which will then pollute the object you read the bits into, eventually leading to inscrutable program behavior and a lot of unpleasant debugging.

Now, there are rules that describe when object initialization is guaranteed to take place and when it isn't. Unfortunately, the rules are complicated — too complicated to be worth memorizing, in my opinion. In general, if you're in the C part of C++ (see Item 1) and initialization would probably incur a runtime cost, it's not guaranteed to take place. If you cross into the non-C parts of C++, things sometimes change. This explains why an array (from the C part of C++) isn't necessarily guaranteed to have its contents initialized, but a `vector` (from the STL part of C++) is.

The best way to deal with this seemingly indeterminate state of affairs is to *always* initialize your objects before you use them. For non-member objects of built-in types, you'll need to do this manually. For example:

Click here to view code image

```
int x = 0;                                   // manual
initialization of an int

const char * text = "A C-style string";      // manual
initialization of a
                                             // pointer (see
also Item 3)

double d;                                    // "initialization"
by reading from
std::cin >> d;                               // an input stream
```

For almost everything else, the responsibility for initialization falls on constructors. The rule there is simple: make sure that all constructors initialize everything in the object.

The rule is easy to follow, but it's important not to confuse assignment with initialization. Consider a constructor for a class representing entries in an address book:

Click here to view code image

```
class PhoneNumber { ... };

class ABEntry {                        // ABEntry = "Address
Book Entry"
public:
  ABEntry(const std::string& name, const std::string&
address,
          const std::list<PhoneNumber>& phones);
private:
  std::string theName;
  std::string theAddress;
  std::list<PhoneNumber> thePhones;
  int num TimesConsulted;
};

ABEntry::ABEntry(const std::string& name, const std::string&
address,
                 const std::list<PhoneNumber>& phones)
{
  theName = name;                        // these are all
assignments,
  theAddress = address;                  // not
initializations
  thePhones = phones;
  numTimesConsulted = 0;
}
```

This will yield `ABEntry` objects with the values you expect, but it's still not the best approach. The rules of C++ stipulate that data members of an object are initialized *before* the body of a constructor is entered. Inside the `ABEntry` constructor, `theName`, `theAddress`, and `thePhones` aren't being initialized, they're being *assigned*. Initialization took place earlier — when their default constructors were automatically called prior to entering the body of the `ABEntry` constructor. This isn't true for `numTimesConsulted`, because it's a built-in type. For it, there's no guarantee it was initialized at all prior to its assignment.

A better way to write the `ABEntry` constructor is to use the member initialization list instead of assignments:

Click here to view code image

```
ABEntry::ABEntry(const std::string& name, const std::string&
address,
                 const std::list<PhoneNumber>& phones)
: theName(name),
  theAddress(address),                  // these are now all
initializations
  thePhones(phones),
  numTimesConsulted(0)
{}                                      // the ctor body is
now empty
```

This constructor yields the same end result as the one above, but it will often be more efficient. The assignment-based version first called default constructors to initialize `theName`, `theAddress`, and `thePhones`, then promptly assigned new values on top of the default-constructed ones. All the work performed in those default constructions was therefore wasted. The member initialization list approach avoids that problem, because the arguments in the initialization list are used as constructor arguments for the various data members. In this case, `theName` is copy-constructed from `name`, `theAddress` is copy-constructed from `address`, and `thePhones` is copy-constructed from `phones`. For most types, a single call to a copy constructor is more efficient — sometimes *much* more efficient — than a call to the default constructor followed by a call to the copy assignment operator.

For objects of built-in type like `numTimesConsulted`, there is no difference in cost between initialization and assignment, but for consistency, it's often best to initialize everything via member initialization. Similarly, you can use the member initialization list even when you want to default-construct a data member; just specify nothing as an initialization argument. For example, if `ABEntry` had a constructor taking no parameters, it could be implemented like this:

Click here to view code image

```
ABEntry::ABEntry()
:theName(),                         // call theName's default
ctor;
theAddress(),                      // do the same for
```

```
theAddress;
thePhones(),                       // and for thePhones;
numTimesConsulted(0)               // but explicitly
initialize
{}                                  // numTimesConsulted to
zero
```

Because compilers will automatically call default constructors for data members of user-defined types when those data members have no initializers on the member initialization list, some programmers consider the above approach overkill. That's understandable, but having a policy of always listing every data member on the initialization list avoids having to remember which data members may go uninitialized if they are omitted. Because `numTimesConsulted` is of a built-in type, for example, leaving it off a member initialization list could open the door to undefined behavior.

Sometimes the initialization list *must* be used, even for built-in types. For example, data members that are `const` or are references must be initialized; they can't be assigned (see also Item 5). To avoid having to memorize when data members must be initialized in the member initialization list and when it's optional, the easiest choice is to *always* use the initialization list. It's sometimes required, and it's often more efficient than assignments.

Many classes have multiple constructors, and each constructor has its own member initialization list. If there are many data members and/or base classes, the existence of multiple initialization lists introduces undesirable repetition (in the lists) and boredom (in the programmers). In such cases, it's not unreasonable to omit entries in the lists for data members where assignment works as well as true initialization, moving the assignments to a single (typically private) function that all the constructors call. This approach can be especially helpful if the true initial values for the data members are to be read from a file or looked up in a database. In general, however, true member initialization (via an initialization list) is preferable to pseudo-initialization via assignment.

One aspect of C++ that isn't fickle is the order in which an object's data is initialized. This order is always the same: base classes are initialized before derived classes (see also Item 12), and within a class, data members are initialized in the order in which they are declared. In `ABEntry`, for example, `theName` will always be initialized first, `theAddress` second, `thePhones` third, and `numTimesConsulted` last. This is true even if they are listed in a different order on the member initialization list (something that's unfortunately legal). To avoid reader confusion, as well as the possibility of some truly obscure behavioral bugs, always list members in the initialization list in the same order as they're declared in the class.

Once you've taken care of explicitly initializing non-member objects of built-in types and you've ensured that your constructors initialize their base classes and data members using the member initialization list, there's only one more thing to worry about. That thing is — take a deep breath — the order of initialization of non-local static objects defined in different translation units.

Let's pick that phrase apart bit by bit.

A *static object* is one that exists from the time it's constructed until the end of the program. Stack and heap-based objects are thus excluded. Included are global objects, objects defined at namespace scope, objects declared `static` inside classes, objects declared `static` inside functions, and objects declared `static` at file scope. Static objects inside functions are known as *local static objects* (because they're local to a function), and the other kinds of static objects are known as *non-local static objects*. Static objects are destroyed when the program exits, i.e., their destructors are called when `main` finishes executing.

A *translation unit* is the source code giving rise to a single object file. It's basically a single source file, plus all of its `#include` files.

The problem we're concerned with, then, involves at least two separately compiled source files, each of which contains at least one non-local static object (i.e., an object that's global, at namespace scope, or `static` in a class or at file scope). And the actual problem is this: if initialization of a non-local static object in one translation unit uses a non-local static object in a different translation unit, the object it uses could be uninitialized, because *the relative order of initialization of non-local static objects defined in different translation units is undefined*.

An example will help. Suppose you have a `FileSystem` class that makes files on the Internet look like they're local. Since your class makes the world look like a single file system, you might create a special object at global or namespace scope representing the single file system:

Click here to view code image

```
class FileSystem {                    // from your library’s
header file public:
  ...
  std::size_t numDisks() const;       // one of many member
functions
  ...
};

extern FileSystem tfs;                // declare object for
clients to use
                                      // ("tfs" = "the file
system"); definition // is in some .cpp file in your library
```

A `FileSystem` object is decidedly non-trivial, so use of the `tfs` object before it has been constructed would be disastrous.

Now suppose some client creates a class for directories in a file system. Naturally, their class uses the `tfs` object:

Click here to view code image

```
class Directory {                       // created by library
client
public:
   Directory( params );
  ...
};

Directory::Directory( params )
{
  ...
  std::size_t disks = tfs.numDisks();   // use the tfs object
  ...
}
```

Further suppose this client decides to create a single `Directory` object for temporary files:

```
Directory tempDir( params );           // directory for
temporary files
```

Now the importance of initialization order becomes apparent: unless `tfs` is initialized before `tempDir`, `tempDir`'s constructor will attempt to use `tfs` before it's been initialized. But `tfs` and `tempDir` were created by different people at different times in different source files — they're non-local static objects defined in different translation units. How can you be sure that `tfs` will be initialized before `tempDir`?

You can't. Again, *the relative order of initialization of non-local static objects defined in different translation units is undefined*. There is a reason for this. Determining the "proper" order in which to initialize non-local static objects is hard. Very hard. Unsolvably hard. In its most general form — with multiple translation units and non-local static objects generated through implicit template instantiations (which may themselves arise via implicit template instantiations) — it's not only impossible to determine the right order of initialization, it's typically not even worth looking for special cases where it *is* possible to determine the right order.

Fortunately, a small design change eliminates the problem entirely. All that has to be done is to move each non-local static object into its own function, where it's declared `static`. These functions return references to the objects they contain. Clients then call the functions instead of referring to the objects. In other words, non-local static objects are replaced with *local* static objects. (Aficionados of design patterns will recognize this as a common implementation of the Singleton pattern.[†])

This approach is founded on C++'s guarantee that local static objects are initialized when the object's definition is first encountered during a call to that function. So if you replace direct accesses to non-local static objects with calls to functions that return references to local static objects, you're guaranteed that the references you get back will refer to initialized objects. As a bonus, if you never call a function emulating a non-local static object, you never incur the cost of constructing and destructing the object, something that can't be said for true non-local static objects.

Here's the technique applied to both `tfs` and `tempDir`:

Click here to view code image

```
class FileSystem { ... };            // as before

FileSystem& tfs()                    // this replaces the tfs
object; it could be
{                                    // static in the
FileSystem class

  static FileSystem fs;             // define and initialize
a local static object
  return fs;                        // return a reference to
it
}

class Directory { ... };            // as before

Directory::Directory( params )      // as before, except
references to tfs are
{                                   // now to tfs()
  ...
  std::size_t disks = tfs().numDisks();
  ...
}

Directory& tempDir()                // this replaces the
tempDir object; it
{                                   // could be static in the
Directory class

  static Directory td; ( params );              //
define/initialize local static object
  return td;                        // return reference to it
}
```

Clients of this modified system program exactly as they used to, except they now refer to `tfs()` and `tempDir()` instead of `tfs` and `tempDir`. That is, they use functions returning references to objects instead of using the objects themselves.

The reference-returning functions dictated by this scheme are always simple: define and initialize a local static object on line 1, return it on line 2. This simplicity makes them excellent candidates for inlining, especially if they're called frequently (see Item 30). On the other hand, the fact that these functions contain static objects makes them problematic in multithreaded systems. Then again, any kind of non-`const` static object — local or non-local — is trouble waiting to happen in the presence of multiple threads. One way to deal with such trouble is to manually invoke all the reference-returning functions during the single-threaded startup portion of the program. This eliminates initialization-related race conditions.

Of course, the idea of using reference-returning functions to prevent initialization order problems is dependent on there being a reasonable initialization order for your objects in the first place. If you have a system where object A must be initialized before object B, but A's initialization is dependent on B's having already been initialized, you are going to have problems, and frankly, you deserve them. If you steer clear of such pathological scenarios, however, the approach described here should serve you nicely, at least in single-threaded applications.

To avoid using objects before they're initialized, then, you need to do only three things. First, manually initialize non-member objects of built-in types. Second, use member initialization lists to initialize all parts of an object. Finally, design around the initialization order uncertainty that afflicts non-local static objects defined in separate translation units.

Things to Remember

- Manually initialize objects of built-in type, because C++ only sometimes initializes them itself.
- In a constructor, prefer use of the member initialization list to assignment inside the body of the constructor. List data members in the initialization list in the same order they're declared in the class.
- Avoid initialization order problems across translation units by replacing non-local static objects with local static objects.

2. Constructors, Destructors, and Assignment Operators

Almost every class you write will have one or more constructors, a destructor, and a copy assignment operator. Little wonder. These are your bread-and-butter functions, the ones that control the fundamental operations of bringing a new object into existence and making sure it's initialized, getting rid of an object and making sure it's properly cleaned up, and giving an object a new value. Making mistakes in these functions will lead to far-reaching — and unpleasant — repercussions throughout your classes, so it's vital that you get them right. In this chapter, I offer guidance on putting together the functions that comprise the backbone of well-formed classes.

Item 5: Know what functions C++ silently writes and calls

When is an empty class not an empty class? When C++ gets through with it. If you don't declare them yourself, compilers will declare their own versions of a copy constructor, a copy assignment operator, and a destructor. Furthermore, if you declare no constructors at all, compilers will also declare a default constructor for you. All these functions will be both `public` and `inline` (see Item 30). As a result, if you write

```
class Empty{};
```

it's essentially the same as if you'd written this:

Click here to view code image

```
class Empty {
public:
  Empty() { ... }                             // default
constructor
  Empty(const Empty& rhs) { ... }             // copy
constructor

  ~Empty() { ... }                            // destructor —
see below
```

```
                                                           // for whether
it's virtual

  Empty& operator=(const Empty& rhs) { ... } // copy
assignment operator
};
```

These functions are generated only if they are needed, but it doesn't take much to need them. The following code will cause each function to be generated:

Click here to view code image

```
Empty e1;                                         // default
constructor;
                                                  // destructor

Empty e2(e1);                                     // copy constructor

e2 = e1;                                          // copy assignment
operator
```

Given that compilers are writing functions for you, what do the functions do? Well, the default constructor and the destructor primarily give compilers a place to put "behind the scenes" code such as invocation of constructors and destructors of base classes and non-static data members. Note that the generated destructor is non-virtual (see Item 7) unless it's for a class inheriting from a base class that itself declares a virtual destructor (in which case the function's virtualness comes from the base class).

As for the copy constructor and the copy assignment operator, the compiler-generated versions simply copy each non-static data member of the source object over to the target object. For example, consider a `NamedObject` template that allows you to associate names with objects of type `T`:

Click here to view code image

```
template<typename T>
class NamedObject {
public:
  NamedObject(const char *name, const T& value);
  NamedObject(const std::string& name, const T& value);
  ...

private:
  std::string nameValue;
  T objectValue;
};
```

Because a constructor is declared in `NamedObject`, compilers won't generate a default constructor. This is important. It means that if you've carefully engineered a class to require constructor arguments, you don't have to worry about compilers overriding your decision by blithely adding a constructor that takes no arguments.

`NamedObject` declares neither copy constructor nor copy assignment operator, so compilers will generate those functions (if they are needed). Look, then, at this use of the copy constructor:

Click here to view code image

```
NamedObject<int> no1("Smallest Prime Number", 2);

NamedObject<int> no2(no1);                  // calls copy
constructor
```

The copy constructor generated by compilers must initialize `no2.nameValue` and `no2.objectValue` using `no1.nameValue` and `no1.objectValue`, respectively. The type of `nameValue` is `string`, and the standard `string` type has a copy constructor, so `no2.nameValue` will be initialized by calling the `string` copy constructor with `no1.nameValue` as its argument. On the other hand, the type of `NamedObject<int>::objectValue` is `int` (because `T` is `int` for this template instantiation), and `int` is a built-in type, so `no2.objectValue` will be initialized by copying the bits in `no1.objectValue`.

The compiler-generated copy assignment operator for `NamedObject<int>` would behave essentially the same way, but in general, compiler-generated copy assignment operators behave as I've described only when the resulting code is both legal and has a reasonable chance of making sense. If either of these tests fails, compilers will refuse to generate an `operator=` for your class.

For example, suppose `NamedObject` were defined like this, where `nameValue` is a *reference* to a string and `objectValue` is a *const* `T`:

Click here to view code image

```
template<typename T>
class NamedObject {
public:
  // this ctor no longer takes a const name, because
nameValue
  // is now a reference-to-non-const string. The char*
constructor
  // is gone, because we must have a string to refer to.
  NamedObject(std::string& name, const T& value);

  ...                               // as above, assume no
                                    // operator= is declared
private:
  std::string& nameValue;           // this is now a
reference
  const T objectValue;              // this is now const
};
```

Now consider what should happen here:

Click here to view code image

```
std::string newDog("Persephone");
std::string oldDog("Satch");

NamedObject<int> p(newDog, 2);               // when I
originally wrote this, our
                                             // dog
Persephone was about to
                                             // have her
second birthday

NamedObject<int> s(oldDog, 36);              // the family
dog Satch (from my
                                             // childhood)
would be 36 if she
                                             // were still
alive

p = s;                                       // what should
happen to
                                             // the data
members in p?
```

Before the assignment, both `p.nameValue` and `s.nameValue` refer to `string` objects, though not the same ones. How should the assignment affect `p.nameValue`? After the assignment, should `p.nameValue` refer to the `string` referred to by `s.nameValue`, i.e., should the reference itself be modified? If so, that breaks new ground, because C++ doesn't provide a way to make a reference refer to a different object. Alternatively, should the `string` object to which `p.nameValue` refers be modified, thus affecting other objects that hold pointers or references to that `string`, i.e., objects not directly involved in the assignment? Is that what the compiler-generated copy assignment operator should do?

Faced with this conundrum, C++ refuses to compile the code. If you want to support copy assignment in a class containing a reference member, you must define the copy assignment operator yourself. Compilers behave similarly for classes containing `const` members (such as `objectValue` in the modified class above). It's not legal to modify `const` members, so compilers are unsure how to treat them during an implicitly generated assignment function. Finally, compilers reject implicit copy assignment operators in derived classes that inherit from base classes declaring the copy assignment operator `private`. After all, compiler-generated copy assignment operators for derived classes are supposed to handle base class parts, too (see Item 12), but in doing so, they certainly can't invoke member functions the derived class has no right to call.

Things to Remember

- Compilers may implicitly generate a class's default constructor, copy constructor, copy assignment operator, and destructor.

Item 6: Explicitly disallow the use of compiler-generated functions you do not want

Real estate agents sell houses, and a software system supporting such agents would naturally have a class representing homes for sale:

```
class HomeForSale { ... };
```

As every real estate agent will be quick to point out, every property is unique — no two are exactly alike. That being the case, the idea of making a *copy* of a `HomeForSale` object makes little sense. How can you copy something that's inherently unique? You'd thus like attempts to copy `HomeForSale` objects to not compile:

Click here to view code image

```
HomeForSale h1;
HomeForSale h2;

HomeForSale h3(h1);                  // attempt to copy h1 —
should
                                     // not compile!

h1 = h2;                             // attempt to copy h2 —
should
                                     // not compile!
```

Alas, preventing such compilation isn't completely straightforward. Usually, if you don't want a class to support a particular kind of functionality, you simply don't declare the function that would provide it. This strategy doesn't work for the copy constructor and copy assignment operator, because, as Item 5 points out, if you don't declare them and somebody tries to call them, compilers declare them for you.

This puts you in a bind. If you don't declare a copy constructor or a copy assignment operator, compilers may generate them for you. Your class thus supports copying. If, on the other hand, you do declare these functions, your class still supports copying. But the goal here is to *prevent* copying!

The key to the solution is that all the compiler generated functions are public. To prevent these functions from being generated, you must declare them yourself, but there is nothing that requires that *you* declare them public. Instead, declare the copy constructor and the copy assignment operator *private*. By declaring a member function explicitly, you prevent compilers from generating their own version, and by making the function `private`, you keep people from calling it.

Mostly. The scheme isn't foolproof, because member and friend functions can still call your private functions. *Unless*, that is, you are clever enough not to *define* them. Then if somebody inadvertently calls one, they'll get an error at link-time. This trick — declaring member functions `private` and deliberately not implementing them — is so well established, it's used to prevent copying in several classes in C++'s iostreams library. Take a look, for example, at the definitions of `ios_base`, `basic_ios`, and `sentry` in your standard library implementation. You'll find that in each case, both the copy constructor and the copy assignment operator are declared `private` and are not defined.

Applying the trick to `HomeForSale` is easy:

Click here to view code image

```
class HomeForSale {
public:
  ...

private:
  ...
  HomeForSale(const HomeForSale&);            // declarations
only
  HomeForSale& operator=(const HomeForSale&);
};
```

You'll note that I've omitted the names of the functions' parameters. This isn't required, it's just a common convention. After all, the functions will never be implemented, much less used, so what's the point in specifying parameter names?

With the above class definition, compilers will thwart client attempts to copy `HomeForSale` objects, and if you inadvertently try to do it in a member or a friend function, the linker will complain.

It's possible to move the link-time error up to compile time (always a good thing — earlier error detection is better than later) by declaring the copy constructor and copy assignment operator `private` not in `HomeForSale` itself, but in a base class specifically designed to prevent copying. The base class is simplicity itself:

Click here to view code image

```
class Uncopyable {
protected:                                   // allow
construction
  Uncopyable() {}                            // and
destruction of
  ~Uncopyable() {}                           // derived
objects...
private:
  Uncopyable(const Uncopyable&);             // ...but
prevent copying
  Uncopyable& operator=(const Uncopyable&);
};
```

To keep `HomeForSale` objects from being copied, all we have to do now is inherit from `Uncopyable`:

Click here to view code image

```
class HomeForSale: private Uncopyable {      // class no
longer
  ...                                        // declares copy
ctor or
};                                           // copy assign.
operator
```

This works, because compilers will try to generate a copy constructor and a copy assignment operator if anybody — even a member or friend function — tries to copy a `HomeForSale` object. As Item 12 explains, the compiler-generated versions of these functions will try to call their base class counterparts, and those calls will be rejected, because the copying operations are private in the base class.

The implementation and use of `Uncopyable` include some subtleties, such as the fact that inheritance from `Uncopyable` needn't be public (see Items 32 and 39) and that `Uncopyable`'s destructor need not be virtual (see Item 7). Because `Uncopyable` contains no data, it's eligible for the empty base class optimization described in Item 39, but because it's a base class, use of this technique could lead to multiple inheritance (see Item 40). Multiple inheritance, in turn, can sometimes disable the empty base class optimization (again, see Item 39). In general, you can ignore these subtleties and just use `Uncopyable` as shown, because it works precisely as advertised. You can also use the version available at Boost (see Item 55). That class is named `noncopyable`. It's a fine class, I just find the name a bit un-, er, *non*natural.

Things to Remember

- To disallow functionality automatically provided by compilers, declare the corresponding member functions `private` and give no implementations. Using a base class like `Uncopyable` is one way to do this.

Item 7: Declare destructors virtual in polymorphic base classes

There are lots of ways to keep track of time, so it would be reasonable to create a `TimeKeeper` base class along with derived classes for different approaches to timekeeping:

Click here to view code image

```
class TimeKeeper {
public:
  TimeKeeper();
  ~TimeKeeper();
  ...
};

class AtomicClock: public TimeKeeper { ... };

class WaterClock: public TimeKeeper { ... };

class WristWatch: public TimeKeeper { ... };
```

Many clients will want access to the time without worrying about the details of how it's calculated, so a *factory function* — a function that returns a base class pointer to a newly-created derived class object — can be used to return a pointer to a timekeeping object:

Click here to view code image

```
TimeKeeper* getTimeKeeper();        // returns a pointer to a
dynamic-
                                    // ally allocated object
of a class
                                    // derived from TimeKeeper
```

In keeping with the conventions of factory functions, the objects returned by `getTimeKeeper` are on the heap, so to avoid leaking memory and other resources, it's important that each returned object be properly `delete`d:

Click here to view code image

```
TimeKeeper *ptk = getTimeKeeper();  // get dynamically
allocated object
                                    // from TimeKeeper
hierarchy

...                                 // use it

delete ptk;                        // release it to avoid
resource leak
```

Item 13 explains that relying on clients to perform the deletion is error-prone, and Item 18 explains how the interface to the factory function can be modified to prevent common client errors, but such concerns are secondary here, because in this Item we address a more fundamental weakness of the code above: even if clients do everything right, there is no way to know how the program will behave.

The problem is that `getTimeKeeper` returns a pointer to a derived class object (e.g., `AtomicClock`), that object is being deleted via a base class pointer (i.e., a `TimeKeeper*` pointer), and the base class (`TimeKeeper`) has a *non-virtual destructor*. This is a recipe for disaster, because C++ specifies that when a derived class object is deleted through a pointer to a base class with a non-virtual destructor, results are undefined. What typically happens at runtime is that the derived part of the object is never destroyed. If `getTimeKeeper` were to return a pointer to an `AtomicClock` object, the `AtomicClock` part of the object (i.e., the data members declared in the `AtomicClock` class) would probably not be destroyed, nor would the `AtomicClock` destructor run. However, the base class part (i.e., the `TimeKeeper` part) typically would be destroyed, thus leading to a curious “partially destroyed” object. This is an excellent way to leak resources, corrupt data structures, and spend a lot of time with a debugger.

Eliminating the problem is simple: give the base class a virtual destructor. Then deleting a derived class object will do exactly what you want. It will destroy the entire object, including all its derived class parts:

Click here to view code image

```
class TimeKeeper {
public:
  TimeKeeper();
  virtual ~TimeKeeper();
  ...
};
TimeKeeper *ptk = getTimeKeeper();

...

delete ptk;                              // now behaves correctly
```

Base classes like `TimeKeeper` generally contain virtual functions other than the destructor, because the purpose of virtual functions is to allow customization of derived class implementations (see Item 34). For example, `TimeKeeper` might have a virtual function, `getCurrentTime`, which would be implemented differently in the various derived classes. Any class with virtual functions should almost certainly have a virtual destructor.

If a class does *not* contain virtual functions, that often indicates it is not meant to be used as a base class. When a class is not intended to be a base class, making the destructor virtual is usually a bad idea. Consider a class for representing points in two-dimensional space:

Click here to view code image

```
class Point {                                // a 2D point
public:
  Point(int xCoord, int yCoord);
  ~Point();

private:
  int x, y;
};
```

If an `int` occupies 32 bits, a `Point` object can typically fit into a 64-bit register. Furthermore, such a `Point` object can be passed as a 64-bit quantity to functions written in other languages, such as C or FORTRAN. If `Point`'s destructor is made virtual, however, the situation changes.

The implementation of virtual functions requires that objects carry information that can be used at runtime to determine which virtual functions should be invoked on the object. This information typically takes the form of a pointer called a `vptr` (“virtual table pointer”). The `vptr` points to an array of function pointers called a `vtbl` (“virtual table”); each class with virtual functions has an associated `vtbl`. When a virtual function is invoked on an object, the actual function called is determined by following the object's `vptr` to a `vtbl` and then looking up the appropriate function pointer in the `vtbl`.

The details of how virtual functions are implemented are unimportant. What *is* important is that if the `Point` class contains a virtual function, objects of that type will increase in size. On a 32-bit architecture, they'll go from 64 bits (for the two `int`s) to 96 bits (for the `int`s plus the `vptr`); on a 64-bit architecture, they may go from 64 to 128 bits, because pointers on such architectures are 64 bits in size. Addition of a `vptr` to `Point` will thus increase its size by 50–100%! No longer can `Point` objects fit in a 64-bit register. Furthermore, `Point` objects in C++ can no longer look like the same structure declared in another language such as C, because their foreign language counterparts will lack the `vptr`. As a result, it is no longer possible to pass `Point`s to and from functions written in other languages unless you explicitly compensate for the `vptr`, which is itself an implementation detail and hence unportable.

The bottom line is that gratuitously declaring all destructors virtual is just as wrong as never declaring them virtual. In fact, many people summarize the situation this way: declare a virtual destructor in a class if and only if that class contains at least one virtual function.

It is possible to get bitten by the non-virtual destructor problem even in the complete absence of virtual functions. For example, the standard `string` type contains no virtual functions, but misguided programmers sometimes use it as a base class anyway:

Click here to view code image

```
class SpecialString: public std::string {   // bad idea!
std::string has a
  ...                                       // non-virtual
destructor
};
```

At first glance, this may look innocuous, but if anywhere in an application you somehow convert a pointer-to-`SpecialString` into a pointer-to- `string` and you then use `delete` on the `string` pointer, you are instantly transported to the realm of undefined behavior:

Click here to view code image

```
SpecialString *pss =   new SpecialString("Impending Doom");
```

```
std::string *ps;

...

ps = pss;                                // SpecialString* ⇒
std::string*

...

delete ps;                               // undefined! In
practice,
                                         // *ps's
SpecialString resources
                                         // will be leaked,
because the
                                         // SpecialString
destructor won't
                                         // be called.
```

The same analysis applies to any class lacking a virtual destructor, including all the STL container types (e.g., `vector`, `list`, `set`, `tr1::unordered_map` (see Item 54), etc.). If you're ever tempted to inherit from a standard container or any other class with a non-virtual destructor, resist the temptation! (Unfortunately, C++ offers no derivation-prevention mechanism akin to Java's `final` classes or C#'s `sealed` classes.)

Occasionally it can be convenient to give a class a pure virtual destructor. Recall that pure virtual functions result in *abstract* classes — classes that can't be instantiated (i.e., you can't create objects of that type). Sometimes, however, you have a class that you'd like to be abstract, but you don't have any pure virtual functions. What to do? Well, because an abstract class is intended to be used as a base class, and because a base class should have a virtual destructor, and because a pure virtual function yields an abstract class, the solution is simple: declare a pure virtual destructor in the class you want to be abstract. Here's an example:

Click here to view code image

```
class AWOV {                             // AWOV = "Abstract
w/o Virtuals"
public:
  virtual ~AWOV() = 0;                   // declare pure
virtual destructor
};
```

This class has a pure virtual function, so it's abstract, and it has a virtual destructor, so you won't have to worry about the destructor problem. There is one twist, however: you must provide a *definition* for the pure virtual destructor:

```
AWOV::~AWOV() {}                     // definition of pure
virtual    dtor
```

The way destructors work is that the most derived class's destructor is called first, then the destructor of each base class is called. Compilers will generate a call to `~AWOV` from its derived classes' destructors, so you have to be sure to provide a body for the function. If you don't, the linker will complain.

The rule for giving base classes virtual destructors applies only to *polymorphic* base classes — to base classes designed to allow the manipulation of derived class types through base class interfaces. `TimeKeeper` is a polymorphic base class, because we expect to be able to manipulate `AtomicClock` and `WaterClock` objects, even if we have only `TimeKeeper` pointers to them.

Not all base classes are designed to be used polymorphically. Neither the standard `string` type, for example, nor the STL container types are designed to be base classes at all, much less polymorphic ones. Some classes are designed to be used as base classes, yet are not designed to be used polymorphically. Such classes — examples include `Uncopyable` from Item 6 and `input_iterator_tag` from the standard library (see Item 47) — are not designed to allow the manipulation of derived class objects via base class interfaces. As a result, they don't need virtual destructors.

Things to Remember

- Polymorphic base classes should declare virtual destructors. If a class has any virtual functions, it should have a virtual destructor.
- Classes not designed to be base classes or not designed to be used polymorphically should not declare virtual destructors.

Item 8: Prevent exceptions from leaving destructors

C++ doesn't prohibit destructors from emitting exceptions, but it certainly discourages the practice. With good reason. Consider:

Click here to view code image

```
class Widget {
public:
  ...
  ~Widget() { ... }             // assume this might emit an
exception
};

void doSomething()
{
  std::vector<Widget> v;
  ...
}                                   // v is automatically
destroyed here
```

When the `vector v` is destroyed, it is responsible for destroying all the `Widget`s it contains. Suppose `v` has ten `Widget`s in it, and during destruction of the first one, an exception is thrown. The other nine `Widget`s still have to be destroyed (otherwise any resources they hold would be leaked), so `v` should invoke their destructors. But suppose that during those calls, a second `Widget` destructor throws an exception. Now there are two simultaneously active exceptions, and that's one too many for C++. Depending on the precise conditions under which such pairs of simultaneously active exceptions arise, program execution either terminates or yields undefined behavior. In this example, it yields undefined behavior. It would yield equally undefined behavior using any other standard library container (e.g., `list`, `set`), any container in TR1 (see Item 54), or even an array. Not that containers or arrays are required to get into trouble. Premature program termination or undefined behavior can result from destructors emitting exceptions even without using containers and arrays. C++ does *not* like destructors that emit exceptions!

That's easy enough to understand, but what should you do if your destructor needs to perform an operation that may fail by throwing an exception? For example, suppose you're working with a class for database connections:

Click here to view code image

```
class DBConnection {
```

```
public:
  ...

  static DBConnection create();        // function to return
                                       // DBConnection
objects; params
                                       // omitted for
simplicity

  void close();                        // close connection;
throw an
};                                     // exception if
closing fails
```

To ensure that clients don't forget to call `close` on `DBConnection` objects, a reasonable idea would be to create a resource-managing class for `DBConnection` that calls `close` in its destructor. Such resource-managing classes are explored in detail in Chapter 3, but here, it's enough to consider what the destructor for such a class would look like:

Click here to view code image

```
class DBConn {                          // class to manage
DBConnection
public:                                 // objects
  ...
  ~DBConn()                            // make sure database
connections
  {                                     // are always closed
   db.close();
   }
private:
  DBConnection db;
};
```

That allows clients to program like this:

Click here to view code image

```
{                                       // open a block

   DBConn dbc(DBConnection::create());  // create
DBConnection object
                                        // and turn it over
to a DBConn
                                        // object to manage

...                                    // use the
DBConnection object
                                        // via the DBConn
interface

}                                       // at end of block,
the DBConn
                                        // object is
destroyed, thus
                                        // automatically
calling close on
                                        // the DBConnection
object
```

This is fine as long as the call to `close` succeeds, but if the call yields an exception, `DBConn`'s destructor will propagate that exception, i.e., allow it to leave the destructor. That's a problem, because destructors that throw mean trouble.

There are two primary ways to avoid the trouble. `DBConn`'s destructor could:

- **Terminate the program** if `close` throws, typically by calling `abort`:

Click here to view code image

```
DBConn::~DBConn()
{
try { db.close(); }
catch (...) {
   make log entry that the call to close failed;
   std::abort();
}
}
```

This is a reasonable option if the program cannot continue to run after an error is encountered during destruction. It has the advantage that if allowing the exception to propagate from the destructor would lead to undefined behavior, this prevents that from happening. That is, calling `abort` may forestall undefined behavior.

- **Swallow the exception** arising from the call to `close`:

Click here to view code image

```
DBConn::~DBConn()
{
try { db.close(); }
catch (...) {
      make log entry that the call to close failed;
}
}
```

In general, swallowing exceptions is a bad idea, because it suppresses important information — *something failed!* Sometimes, however, swallowing exceptions is preferable to running the risk of premature program termination or undefined behavior. For this to be a viable option, the program must be able to reliably continue execution even after an error has been encountered and ignored.

Neither of these approaches is especially appealing. The problem with both is that the program has no way to react to the condition that led to `close` throwing an exception in the first place.

A better strategy is to design `DBConn`'s interface so that its clients have an opportunity to react to problems that may arise. For example, `DBConn` could offer a `close` function itself, thus giving clients a chance to handle exceptions arising from that operation. It could also keep track of whether its `DBConnection` had been `close`d, closing it itself in the destructor if not. That would prevent a connection from leaking. If the call to `close` were to fail in the `DBConnection` destructor, however, we'd be back to terminating or swallowing:

Click here to view code image

```
class DBConn {
public:
  ...

  void close()                                         // new
function for
  {                                                    // client
use
    db.close();
    closed = true;
  }

  ~DBConn()
  {
    if (!closed) {
        try {                                                  //
close the connection
           db.close();                                          //
if the client didn't
        }
        catch (...) {                                          // if
closing fails,
          make log entry that call to close failed;   // note
that and
          ...                                                   //
terminate or swallow
        }
     }
   }
private:
  DBConnection db;
  bool closed;
};
```

Moving the responsibility for calling `close` from `DBConn`'s destructor to `DBConn`'s client (with `DBConn`'s destructor containing a "backup" call) may strike you as an unscrupulous shift of burden. You might even view it as a violation of Item 18's advice to make interfaces easy to use correctly. In fact, it's neither. If an operation may fail by throwing an exception and there may be a need to handle that exception, the exception *has to come from some non-destructor function*. That's because destructors that emit exceptions are dangerous, always running the risk of premature program termination or undefined behavior. In this example, telling clients to call `close` themselves doesn't impose a burden on them; it gives them an opportunity to deal with errors they would otherwise have no chance to react to. If they don't find that opportunity useful (perhaps because they believe that no error will really occur), they can ignore it, relying on `DBConn`'s destructor to call `close` for them. If an error occurs at that point — if `close` *does* throw — they're in no position to complain if `DBConn` swallows the exception or terminates the program. After all, they had first crack at dealing with the problem, and they chose not to use it.

Things to Remember

- Destructors should never emit exceptions. If functions called in a destructor may throw, the destructor should catch any exceptions, then swallow them or terminate the program.

- If class clients need to be able to react to exceptions thrown during an operation, the class should provide a regular (i.e., non-destructor) function that performs the operation.

Item 9: Never call virtual functions during construction or destruction

I'll begin with the recap: you shouldn't call virtual functions during construction or destruction, because the calls won't do what you think, and if they did, you'd still be unhappy. If you're a recovering Java or C# programmer, pay close attention to this Item, because this is a place where those languages zig, while C++ zags.

Suppose you've got a class hierarchy for modeling stock transactions, e.g., buy orders, sell orders, etc. It's important that such transactions be auditable, so each time a transaction object is created, an appropriate entry needs to be created in an audit log. This seems like a reasonable way to approach the problem:

Click here to view code image

```
class Transaction {                               // base
class for all
public:                                           //
transactions
  Transaction();

  virtual void logTransaction() const = 0;       // make
type-dependent
                                                 // log entry
  ...
};

Transaction::Transaction()                        //
implementation of
{                                                 // base
class ctor
  ...
  logTransaction();                               // as final
action, log this
}                                                 //
```

```
transaction
class BuyTransaction: public Transaction {        // derived
class
public:
  virtual void logTransaction() const;          // how to log
trans-
                                                // actions of
this type
  ...
};
class SellTransaction: public Transaction {      // derived
class
public:
virtual void logTransaction() const;           // how to log
trans-
                                                // actions of
this type
  ...
};
```

Consider what happens when this code is executed:

```
BuyTransaction b;
```

Clearly a `BuyTransaction` constructor will be called, but first, a `Transaction` constructor must be called; base class parts of derived class objects are constructed before derived class parts are. The last line of the `Transaction` constructor calls the virtual function `logTransaction`, but this is where the surprise comes in. The version of `logTransaction` that's called is the one in `Transaction`, *not* the one in `BuyTransaction` — even though the type of object being created is `BuyTransaction`. During base class construction, virtual functions never go down into derived classes. Instead, the object behaves as if it were of the base type. Informally speaking, during base class construction, virtual functions aren't.

There's a good reason for this seemingly counterintuitive behavior. Because base class constructors execute before derived class constructors, derived class data members have not been initialized when base class constructors run. If virtual functions called during base class construction went down to derived classes, the derived class functions would almost certainly refer to local data members, but those data members would not yet have been initialized. That would be a non-stop ticket to undefined behavior and late-night debugging sessions. Calling down to parts of an object that have not yet been initialized is inherently dangerous, so C++ gives you no way to do it.

It's actually more fundamental than that. During base class construction of a derived class object, the type of the object *is* that of the base class. Not only do virtual functions resolve to the base class, but the parts of the language using runtime type information (e.g., `dynamic_cast` (see Item 27) and `typeid`) treat the object as a base class type. In our example, while the `Transaction` constructor is running to initialize the base class part of a `BuyTransaction` object, the object is of type `Transaction`. That's how every part of C++ will treat it, and the treatment makes sense: the `BuyTransaction`-specific parts of the object haven't been initialized yet, so it's safest to treat them as if they didn't exist. An object doesn't become a derived class object until execution of a derived class constructor begins.

The same reasoning applies during destruction. Once a derived class destructor has run, the object's derived class data members assume undefined values, so C++ treats them as if they no longer exist. Upon entry to the base class destructor, the object becomes a base class object, and all parts of C++ — virtual functions, `dynamic_cast`s, etc., — treat it that way.

In the example code above, the `Transaction` constructor made a direct call to a virtual function, a clear and easy-to-see violation of this Item's guidance. The violation is so easy to see, some compilers issue a warning about it. (Others don't. See Item 53 for a discussion of warnings.) Even without such a warning, the problem would almost certainly become apparent before runtime, because the `logTransaction` function is pure virtual in `Transaction`. Unless it had been defined (unlikely, but possible — see Item 34), the program wouldn't link: the linker would be unable to find the necessary implementation of `Transaction::logTransaction`.

It's not always so easy to detect calls to virtual functions during construction or destruction. If `Transaction` had multiple constructors, each of which had to perform some of the same work, it would be good software engineering to avoid code replication by putting the common initialization code, including the call to `logTransaction`, into a private non-virtual initialization function, say, `init`:

Click here to view code image

```
class Transaction {
```

```
public:
  Transaction()
  { init(); }                                    // call to
non-virtual...

  virtual void logTransaction() const = 0;
  ...

private:
  void init()
  {
    ...
    logTransaction();                            // ...that
calls a virtual!
  }
};
```

This code is conceptually the same as the earlier version, but it's more insidious, because it will typically compile and link without complaint. In this case, because `logTransaction` is pure virtual in `Transaction`, most runtime systems will abort the program when the pure virtual is called (typically issuing a message to that effect). However, if `logTransaction` were a "normal" virtual function (i.e., not pure virtual) with an implementation in `Transaction`, that version would be called, and the program would merrily trot along, leaving you to figure out why the wrong version of `logTransaction` was called when a derived class object was created. The only way to avoid this problem is to make sure that none of your constructors or destructors call virtual functions on the object being created or destroyed and that all the functions they call obey the same constraint.

But how *do* you ensure that the proper version of `logTransaction` is called each time an object in the `Transaction` hierarchy is created? Clearly, calling a virtual function on the object from the `Transaction` constructor(s) is the wrong way to do it.

There are different ways to approach this problem. One is to turn `logTransaction` into a non-virtual function in `Transaction`, then require that derived class constructors pass the necessary log information to the `Transaction` constructor. That function can then safely call the non-virtual `logTransaction`. Like this:

Click here to view code image

```
class Transaction {
public:
  explicit Transaction(const std::string& logInfo);

  void logTransaction(const std::string& logInfo) const;   //
now a non-
                                                           //
virtual func
  ...
};

Transaction::Transaction(const std::string& logInfo)
{
  ...
  logTransaction(logInfo);                                //
now a non-
}                                                         //
virtual call

class BuyTransaction: public Transaction {
public:
BuyTransaction( parameters )
: Transaction(createLogString( parameters ))             //
pass log info
  { ... }                                                 //
to base class
   ...                                                    //
constructor

private:
  static std::string createLogString( parameters );
};
```

In other words, since you can't use virtual functions to call down from base classes during construction, you can compensate by having derived classes pass necessary construction information up to base class constructors instead.

In this example, note the use of the (private) static function `createLogString` in `BuyTransaction`. Using a helper function to create a value to pass to a base class constructor is often more convenient (and more readable) than going through contortions in the member initialization list to give the base class what it needs. By making the function static, there's no danger of accidentally referring to the nascent `BuyTransaction` object's as-yet-uninitialized data members. That's important, because the fact that those data members will be in an undefined state is why calling virtual functions during base class construction and destruction doesn't go down into derived classes in the first place.

Things to Remember

- Don't call virtual functions during construction or destruction, because such calls will never go to a more derived class than that of the currently executing constructor or destructor.

Item 10: Have assignment operators return a reference to `*this`

One of the interesting things about assignments is that you can chain them together:

```
int x, y, z;

x = y = z = 15;                              // chain of
assignments
```

Also interesting is that assignment is right-associative, so the above assignment chain is parsed like this:

```
x = (y = (z = 15));
```

Here, 15 is assigned to `z`, then the result of that assignment (the updated `z`) is assigned to `y`, then the result of that assignment (the updated `y`) is assigned to `x`.

The way this is implemented is that assignment returns a reference to its left-hand argument, and that's the convention you should follow when you implement assignment operators for your classes:

Click here to view code image

```
class Widget {
public:
  ...
Widget& operator=(const Widget& rhs)   // return type is a
reference to
{                                      // the current class
  ...
  return *this;                        // return the left-
hand object
  }
  ...
};
```

This convention applies to all assignment operators, not just the standard form shown above. Hence:

Click here to view code image

```
class Widget {
public:
  ...
  Widget& operator+=(const Widget& rhs)   // the convention
applies to
  {                                      // +=, -=, *=, etc.
   ...
   return *this;
  }
   Widget& operator=(int rhs)            // it applies even
if the
   {                                     // operator's
parameter type
      ...                                // is unconventional
      return *this;
   }
   ...
};
```

This is only a convention; code that doesn't follow it will compile. However, the convention is followed by all the built-in types as well as by all the types in (or soon to be in — see Item 54) the standard library (e.g., `string`, `vector`, `complex`, `tr1::shared_ptr`, etc.). Unless you have a good reason for doing things differently, don't.

Things to Remember

- Have assignment operators return a reference to `*this`.

Item 11: Handle assignment to self in `operator=`

An assignment to self occurs when an object is assigned to itself:

Click here to view code image

```
class Widget { ... };

Widget w;
...

w = w;                                   // assignment to
self
```

This looks silly, but it's legal, so rest assured that clients will do it. Besides, assignment isn't always so recognizable. For example,

```
a[i] = a[j];                                      //
potential assignment to self
```

is an assignment to self if `i` and `j` have the same value, and

```
*px = *py;                                        //
potential assignment to self
```

is an assignment to self if `px` and `py` happen to point to the same thing. These less obvious assignments to self are the result of *aliasing*: having more than one way to refer to an object. In general, code that operates on references or pointers to multiple objects of the same type needs to consider that the objects might be the same. In fact, the two objects need not even be declared to be of the same type if they're from the same hierarchy, because a base class reference or pointer can refer or point to an object of a derived class type:

Click here to view code image

```
class Base { ... };

class Derived: public Base { ... };

void doSomething(const Base& rb,                   // rb and
*pd might actually be
                 Derived* pd);                      // the
```

```
same object
```

If you follow the advice of Items 13 and 14, you'll always use objects to manage resources, and you'll make sure that the resource-managing objects behave well when copied. When that's the case, your assignment operators will probably be self-assignment-safe without your having to think about it. If you try to manage resources yourself, however (which you'd certainly have to do if you were writing a resource-managing class), you can fall into the trap of accidentally releasing a resource before you're done using it. For example, suppose you create a class that holds a raw pointer to a dynamically allocated bitmap:

Click here to view code image

```
class Bitmap { ... };

class Widget {
  ...

private:
  Bitmap *pb;                                         // ptr to a
heap-allocated object
};
```

Here's an implementation of `operator=` that looks reasonable on the surface but is unsafe in the presence of assignment to self. (It's also not exception-safe, but we'll deal with that in a moment.)

Click here to view code image

```
Widget&
Widget::operator=(const Widget& rhs)                  // unsafe
impl. of operator=
{
  delete pb;                                          // stop
using current bitmap
  pb = new Bitmap(*rhs.pb);                           // start
using a copy of rhs's bitmap

  return *this;                                       // see Item
10
}
```

The self-assignment problem here is that inside `operator=`, `*this` (the target of the assignment) and `rhs` could be the same object. When they are, the `delete` not only destroys the bitmap for the current object, it destroys the bitmap for `rhs`, too. At the end of the function, the `Widget` — which should not have been changed by the assignment to self — finds itself holding a pointer to a deleted object![‡]

The traditional way to prevent this error is to check for assignment to self via an *identity test* at the top of `operator=`:

Click here to view code image

```
Widget& Widget::operator=(const Widget& rhs)
{
  if (this == &rhs) return *this;   // identity test: if a
self-assignment,
                                    // do nothing
  delete pb;
  pb = new Bitmap(*rhs.pb);

  return *this;
}
```

This works, but I mentioned above that the previous version of `operator=` wasn't just self-assignment-unsafe, it was also exception-unsafe, and this version continues to have exception trouble. In particular, if the "`new Bitmap`" expression yields an exception (either because there is insufficient memory for the allocation or because `Bitmap`'s copy constructor throws one), the `Widget` will end up holding a pointer to a deleted `Bitmap`. Such pointers are toxic. You can't safely delete them. You can't even safely read them. About the only safe thing you can do with them is spend lots of debugging energy figuring out where they came from.

Happily, making `operator=` exception-safe typically renders it self-assignment-safe, too. As a result, it's increasingly common to deal with issues of self-assignment by ignoring them, focusing instead on achieving exception safety. Item 29 explores exception safety in depth, but in this Item, it suffices to observe that in many cases, a careful ordering of statements can yield exception-safe (and self-assignment-safe) code. Here, for example, we just have to be careful not to delete `pb` until after we've copied what it points to:

Click here to view code image

```
Widget& Widget::operator=(const Widget& rhs)
{
  Bitmap *pOrig = pb;               // remember original pb
  pb = new Bitmap(*rhs.pb);         // make pb point to a
copy of *pb
  delete pOrig;                     // delete the original pb

  return *this;
}
```

Now, if “`new Bitmap`” throws an exception, `pb` (and the `Widget` it's inside of) remains unchanged. Even without the identity test, this code handles assignment to self, because we make a copy of the original bitmap, delete the original bitmap, then point to the copy we made. It may not be the most efficient way to handle self-assignment, but it does work.

If you're concerned about efficiency, you could put the identity test back at the top of the function. Before doing that, however, ask yourself how often you expect self-assignments to occur, because the test isn't free. It makes the code (both source and object) a bit bigger, and it introduces a branch into the flow of control, both of which can decrease runtime speed. The effectiveness of instruction prefetching, caching, and pipelining can be reduced, for example.

An alternative to manually ordering statements in `operator=` to make sure the implementation is both exception- and self-assignment-safe is to use the technique known as “copy and swap.” This technique is closely associated with exception safety, so it's described in Item 29. However, it's a common enough way to write `operator=` that it's worth seeing what such an implementation often looks like:

Click here to view code image

```
class Widget {
  ...
  void swap(Widget& rhs);    // exchange *this's and rhs's
data;
  ...                        // see Item 29 for details
};

Widget& Widget::operator=(const Widget& rhs)
{
  Widget temp(rhs);                 // make a copy of rhs's data

  swap(temp);                       // swap *this's data with the
copy's
  return *this;
}
```

A variation on this theme takes advantage of the facts that (1) a class's copy assignment operator may be declared to take its argument by value and (2) passing something by value makes a *copy* of it (see Item 20):

Click here to view code image

```
Widget& Widget::operator=(Widget rhs)   // rhs is a copy of
the object
{                                       // passed in — note
pass by val

  swap(rhs);                            // swap *this's data
with
                                        // the copy's

  return *this;
}
```

Personally, I worry that this approach sacrifices clarity at the altar of cleverness, but by moving the copying operation from the body of the function to construction of the parameter, it's a fact that compilers can sometimes generate more efficient code.

Things to Remember

- Make sure `operator=` is well-behaved when an object is assigned to itself. Techniques include comparing addresses of source and target objects, careful statement ordering, and copy-and-`swap`.
- Make sure that any function operating on more than one object behaves correctly if two or more of the objects are the same.

Item 12: Copy all parts of an object

In well-designed object-oriented systems that encapsulate the internal parts of objects, only two functions copy objects: the aptly named copy constructor and copy assignment operator. We'll call these the *copying functions*. Item 5 observes that compilers will generate the copying functions, if needed, and it explains that the compiler-generated versions do precisely what you'd expect: they copy all the data of the object being copied.

When you declare your own copying functions, you are indicating to compilers that there is something about the default implementations you don't like. Compilers seem to take offense at this, and they retaliate in a curious fashion: they don't tell you when your implementations are almost certainly wrong.

Consider a class representing customers, where the copying functions have been manually written so that calls to them are logged:

Click here to view code image

```
void logCall(const std::string& funcName);          // make a
log entry

class Customer {
public:
  ...
  Customer(const Customer& rhs);
  Customer& operator=(const Customer& rhs);
  ...

private:
  std::string name;
};

Customer::Customer(const Customer& rhs)
: name(rhs.name)                                 // copy
rhs's data
{
  logCall("Customer copy constructor");
}

Customer& Customer::operator=(const Customer& rhs)
{
  logCall("Customer copy assignment operator");

  name = rhs.name;                               // copy
rhs's data
```

```
  return *this;                                    // see Item
10
}
```

Everything here looks fine, and in fact everything is fine — until another data member is added to `Customer`:

Click here to view code image

```
class Date { ... };         // for dates in time

class Customer {
public:
  ...                       // as before

private:
  std::string name;
  Date lastTransaction;
};
```

At this point, the existing copying functions are performing a *partial copy*: they're copying the customer's `name`, but not its `lastTransaction`. Yet most compilers say nothing about this, not even at maximal warning level (see also Item 53). That's their revenge for your writing the copying functions yourself. You reject the copying functions they'd write, so they don't tell you if your code is incomplete. The conclusion is obvious: if you add a data member to a class, you need to make sure that you update the copying functions, too. (You'll also need to update all the constructors (see Items 4 and 45) as well as any non-standard forms of `operator=` in the class (Item 10 gives an example). If you forget, compilers are unlikely to remind you.)

One of the most insidious ways this issue can arise is through inheritance. Consider:

Click here to view code image

```
class PriorityCustomer: public Customer {                   //
a derived class
public:
   ...
   PriorityCustomer(const PriorityCustomer& rhs);
   PriorityCustomer& operator=(const PriorityCustomer& rhs);
   ...
private:
   int priority;
};

PriorityCustomer::PriorityCustomer(const PriorityCustomer&
rhs)
: priority(rhs.priority)
{
  logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
  logCall("PriorityCustomer copy assignment operator");

  priority = rhs.priority;

  return *this;
}
```

`PriorityCustomer`'s copying functions look like they're copying everything in `PriorityCustomer`, but look again. Yes, they copy the data member that `PriorityCustomer` declares, but every `PriorityCustomer` also contains a copy of the data members it inherits from `Customer`, and those data members are not being copied at all! `PriorityCustomer`'s copy constructor specifies no arguments to be passed to its base class constructor (i.e., it makes no mention of `Customer` on its member initialization list), so the `Customer` part of the `PriorityCustomer` object will be initialized by the `Customer` constructor taking no arguments — by the default constructor. (Assuming it has one. If not, the code won't compile.) That constructor will perform a *default* initialization for `name` and `lastTransaction`.

The situation is only slightly different for `PriorityCustomer`'s copy assignment operator. It makes no attempt to modify its base class data members in any way, so they'll remain unchanged.

Any time you take it upon yourself to write copying functions for a derived class, you must take care to also copy the base class parts. Those parts are typically private, of course (see Item 22), so you can't access them directly. Instead, derived class copying functions must invoke their corresponding base class functions:

Click here to view code image

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer&
rhs)
:    Customer(rhs),                    // invoke base class
copy ctor
  priority(rhs.priority)
{
  logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
  logCall("PriorityCustomer copy assignment operator");

  Customer::operator=(rhs);            // assign base class
parts
  priority = rhs.priority;

  return *this;
}
```

The meaning of “copy all parts” in this Item's title should now be clear. When you're writing a copying function, be sure to (1) copy all local data members and (2) invoke the appropriate copying function in all base classes, too.

In practice, the two copying functions will often have similar bodies, and this may tempt you to try to avoid code duplication by having one function call the other. Your desire to avoid code duplication is laudable, but having one copying function call the other is the wrong way to achieve it.

It makes no sense to have the copy assignment operator call the copy constructor, because you'd be trying to construct an object that already exists. This is so nonsensical, there's not even a syntax for it. There are syntaxes that *look* like you're doing it, but you're not; and there are syntaxes that *do* do it in a backwards kind of way, but they corrupt your object under some conditions. So I'm not going to show you any of those syntaxes. Simply accept that having the copy assignment operator call the copy constructor is something you don't want to do.

Trying things the other way around — having the copy constructor call the copy assignment operator — is equally nonsensical. A constructor initializes new objects, but an assignment operator applies only to objects that have already been initialized. Performing an assignment on an object under construction would mean doing something to a not-yet-initialized object that makes sense only for an initialized object. Nonsense! Don't try it.

Instead, if you find that your copy constructor and copy assignment operator have similar code bodies, eliminate the duplication by creating a third member function that both call. Such a function is typically private and is often named `init`. This strategy is a safe, proven way to eliminate code duplication in copy constructors and copy assignment operators.

Things to Remember

- Copying functions should be sure to copy all of an object's data members and all of its base class parts.
- Don't try to implement one of the copying functions in terms of the other. Instead, put common functionality in a third function that both call.

3. Resource Management

A resource is something that, once you're done using it, you need to return to the system. If you don't, bad things happen. In C++ programs, the most commonly used resource is dynamically allocated memory (if you allocate memory and never deallocate it, you've got a memory leak), but memory is only one of many resources you must manage. Other common resources include file descriptors, mutex locks, fonts and brushes in graphical user interfaces (GUIs), database connections, and network sockets. Regardless of the resource, it's important that it be released when you're finished with it.

Trying to ensure this by hand is difficult under any conditions, but when you consider exceptions, functions with multiple return paths, and maintenance programmers modifying software without fully comprehending the impact of their changes, it becomes clear that ad hoc ways of dealing with resource management aren't sufficient.

This chapter begins with a straightforward object-based approach to resource management built on C++'s support for constructors, destructors, and copying operations. Experience has shown that disciplined adherence to this approach can all but eliminate resource management problems. The chapter then moves on to Items dedicated specifically to memory management. These latter Items complement the more general Items that come earlier, because objects that manage memory have to know how to do it properly.

Item 13: Use objects to manage resources.

Suppose we're working with a library for modeling investments (e.g., stocks, bonds, etc.), where the various investment types inherit from a root class `Investment`:

```
class Investment { ... };            // root class of
hierarchy of
                                     // investment types
```

Further suppose that the way the library provides us with specific `Investment` objects is through a factory function (see Item 7):

Click here to view code image

```
Investment* createInvestment();  // return ptr to dynamically
allocated
                                 // object in the Investment
hierarchy;
                                 // the caller must delete it
                                 // (parameters omitted for
simplicity)
```

As the comment indicates, callers of `createInvestment` are responsible for deleting the object that function returns when they are done with it. Consider, then, a function `f` written to fulfill this obligation:

Click here to view code image

```
void f()
{
  Investment *pInv = createInvestment();         // call
factory function

  ...                                            // use pInv

  delete pInv;                                   // release
object
}
```

This looks okay, but there are several ways `f` could fail to delete the investment object it gets from `createInvestment`. There might be a premature `return` statement somewhere inside the "..." part of the function. If such a `return` were executed, control would never reach the `delete` statement. A similar situation would arise if the uses of `createInvestment` and `delete` were in a loop, and the loop was prematurely exited by a `break` or `goto` statement. Finally, some statement inside the "..." might throw an exception. If so, control would again not get to the `delete`. Regardless of how the `delete` were to be skipped, we'd leak not only the memory containing the investment object but also any resources held by that object.

Of course, careful programming could prevent these kinds of errors, but think about how the code might change over time. As the software gets maintained, somebody might add a `return` or `continue` statement without fully grasping the repercussions on the rest of the function's resource management strategy. Even worse, the "..." part of `f` might call a function that never used to throw an exception but suddenly starts doing so after it has been “improved.” Relying on `f` always getting to its `delete` statement simply isn't viable.

To make sure that the resource returned by `createInvestment` is always released, we need to put that resource inside an object whose destructor will automatically release the resource when control leaves `f`. In fact, that's half the idea behind this Item: by putting resources inside objects, we can rely on C++'s automatic destructor invocation to make sure that the resources are released. (We'll discuss the other half of the idea in a moment.)

Many resources are dynamically allocated on the heap, are used only within a single block or function, and should be released when control leaves that block or function. The standard library's `auto_ptr` is tailor-made for this kind of situation. `auto_ptr` is a pointer-like object (a *smart pointer*) whose destructor automatically calls `delete` on what it points to. Here's how to use `auto_ptr` to prevent `f`'s potential resource leak:

Click here to view code image

```
void f()
{
  std::auto_ptr<Investment> pInv(createInvestment());   //
call factory
                                                        //
function

  ...                                                   // use
pInv as
                                                        //
before

}                                                       //
automatically
                                                        //
delete pInv via
                                                        //
auto_ptr's dtor
```

This simple example demonstrates the two critical aspects of using objects to manage resources:

- **Resources are acquired and immediately turned over to resource-managing objects.** Above, the resource returned by `createInvestment` is used to initialize the `auto_ptr` that will manage it. In fact, the idea of using objects to manage resources is often called *Resource Acquisition Is Initialization* (RAII), because it's so common to acquire a resource and initialize a resource-managing object in the same statement. Sometimes acquired resources are *assigned* to resource-managing objects instead of initializing them, but either way, every resource is immediately turned over to a resource-managing object at the time the resource is acquired.

- **Resource-managing objects use their destructors to ensure that resources are released.** Because destructors are called automatically when an object is destroyed (e.g., when an object goes out of scope), resources are correctly released, regardless of how control leaves a block. Things can get tricky when the act of releasing resources can lead to exceptions being thrown, but that's a matter addressed by Item 8, so we'll not worry about it here.

Because an `auto_ptr` automatically deletes what it points to when the `auto_ptr` is destroyed, it's important that there never be more than one `auto_ptr` pointing to an object. If there were, the object would be deleted more than once, and that would put your program on the fast track to undefined behavior. To prevent such problems, `auto_ptr`s have an unusual characteristic: copying them (via copy constructor or copy assignment operator) sets them to null, and the copying pointer assumes sole ownership of the resource!

Click here to view code image

```
std::auto_ptr<Investment>                 // pInv1 points to
the
  pInv1(createInvestment());              // object returned
from
                                          // createInvestment

std::auto_ptr<Investment> pInv2(pInv1);   // pInv2 now points
to the
                                          // object; pInv1 is

now null

pInv1 = pInv2;                            // now pInv1 points
to the
                                          // object, and

pInv2 is null
```

This odd copying behavior, plus the underlying requirement that resources managed by `auto_ptr`s must never have more than one `auto_ptr` pointing to them, means that `auto_ptr`s aren't the best way to manage all dynamically allocated resources. For example, STL containers require that their contents exhibit “normal” copying behavior, so containers of `auto_ptr` aren't allowed.

An alternative to `auto_ptr` is a *reference-counting smart pointer* (RCSP). An RCSP is a smart pointer that keeps track of how many objects point to a particular resource and automatically deletes the resource when nobody is pointing to it any longer. As such, RCSPs offer behavior that is similar to that of garbage collection. Unlike garbage collection, however, RCSPs can't break cycles of references (e.g., two otherwise unused objects that point to one another).

TR1's `tr1::shared_ptr` (see Item 54) is an RCSP, so you could write `f` this way:

Click here to view code image

```
void f()
{
  ...

  std::tr1::shared_ptr<Investment>
    pInv(createInvestment());             // call factory
function

  ...                                     // use pInv as
before

}                                         // automatically
delete
                                          // pInv via
shared_ptr's dtor
```

This code looks almost the same as that employing `auto_ptr`, but copying `shared_ptr`s behaves much more naturally:

Click here to view code image

```
void f()
{
  ...
  std::tr1::shared_ptr<Investment>          // pInv1 points
to the
    pInv1(createInvestment());              // object
returned from
                                            //

createInvestment


  std::tr1::shared_ptr<Investment>          // both
pInv1 and pInv2 now
    pInv2(pInv1);                           // point to the
object

  pInv1 = pInv2;                            // ditto —
nothing has
                                            // changed
  ...
}                                           // pInv1 and
pInv2 are
                                            // destroyed, and
the
                                            // object they
point to is
                                            // automatically
deleted
```

Because copying `tr1::shared_ptr`s works “as expected,” they can be used in STL containers and other contexts where `auto_ptr`'s unorthodox copying behavior is inappropriate.

Don't be misled, though. This Item isn't about `auto_ptr`, `tr1::shared_ptr`, or any other kind of smart pointer. It's about the importance of using objects to manage resources. `auto_ptr` and `tr1::shared_ptr` are just examples of objects that do that. (For more information on `tr1:shared_ptr`, consult Items 14, 18, and 54.)

Both `auto_ptr` and `tr1::shared_ptr` use `delete` in their destructors, not `delete []`. (Item 16 describes the difference.) That means that using `auto_ptr` or `tr1::shared_ptr` with dynamically allocated arrays is a bad idea, though, regrettably, one that will compile:

Click here to view code image

```
std::auto_ptr<std::string>                             // bad idea!
the wrong
  aps(new std::string[10]);                            // delete
form will be used

std::tr1::shared_ptr<int> spi(new int[1024]);          // same
problem
```

You may be surprised to discover that there is nothing like `auto_ptr` or `tr1::shared_ptr` for dynamically allocated arrays in C++, not even in TR1. That's because `vector` and `string` can almost always replace dynamically allocated arrays. If you still think it would be nice to have `auto_ptr`- and `tr1::shared_ptr`-like classes for arrays, look to Boost (see Item 55). There you'll be pleased to find the `boost::scoped_array` and `boost::shared_array` classes that offer the behavior you're looking for.

This Item's guidance to use objects to manage resources suggests that if you're releasing resources manually (e.g., using `delete` other than in a resource-managing class), you're doing something wrong. Pre-canned resource-managing classes like `auto_ptr and tr1::shared_ptr` often make following this Item's advice easy, but sometimes you're using a resource where these pre-fab classes don't do what you need. When that's the case, you'll need to craft your own resource-managing classes. That's not terribly difficult to do, but it does involve some subtleties you'll need to consider. Those considerations are the topic of Items 14 and 15.

As a final comment, I have to point out that `createInvestment`'s raw pointer return type is an invitation to a resource leak, because it's so easy for callers to forget to call `delete` on the pointer they get back. (Even if they use an `auto_ptr` or `tr1::shared_ptr` to perform the `delete`, they still have to remember to store `createInvestment`'s return value in a smart pointer object.) Combatting that problem calls for an interface modification to `createInvestment`, a topic I address in Item 18.

Things to Remember

- To prevent resource leaks, use RAII objects that acquire resources in their constructors and release them in their destructors.
- Two commonly useful RAII classes are `tr1::shared_ptr` and `auto_ptr`. `tr1::shared_ptr` is usually the better choice, because its

behavior when copied is intuitive. Copying an `auto_ptr` sets it to null.

Item 14: Think carefully about copying behavior in resource-managing classes.

Item 13 introduces the idea of *Resource Acquisition Is Initialization* (RAII) as the backbone of resource-managing classes, and it describes how `auto_ptr` and `tr1::shared_ptr` are manifestations of this idea for heap-based resources. Not all resources are heap-based, however, and for such resources, smart pointers like `auto_ptr` and `tr1::shared_ptr` are generally inappropriate as resource handlers. That being the case, you're likely to find yourself needing to create your own resource-managing classes from time to time.

For example, suppose you're using a C API to manipulate mutex objects of type `Mutex` offering functions `lock` and `unlock`:

Click here to view code image

```
void lock(Mutex *pm);               // lock mutex pointed to
by pm

void unlock(Mutex *pm);             // unlock the mutex
```

To make sure that you never forget to unlock a `Mutex` you've locked, you'd like to create a class to manage locks. The basic structure of such a class is dictated by the RAII principle that resources are acquired during construction and released during destruction:

Click here to view code image

```
class Lock {
public:
  explicit Lock(Mutex *pm)
  : mutexPtr(pm)
  { lock(mutexPtr); }                          // acquire
resource

  ~Lock() { unlock(mutexPtr); }                // release
resource

private:
  Mutex *mutexPtr;
};
```

Clients use `Lock` in the conventional RAII fashion:

Click here to view code image

```
Mutex m;                    // define the mutex you need to
use

...

{                           // create block to define
critical section
Lock ml(&m);               // lock the mutex

...                         // perform critical section
operations

}                           // automatically unlock mutex at
end
                            // of block
```

This is fine, but what should happen if a `Lock` object is copied?

Click here to view code image

```
Lock ml1(&m);                      // lock m

Lock ml2(ml1);                     // copy ml1 to ml2—what
should
                                   // happen here?
```

This is a specific example of a more general question, one that every RAII class author must confront: what should happen when an RAII object is copied? Most of the time, you'll want to choose one of the following possibilities:

- **Prohibit copying.** In many cases, it makes no sense to allow RAII objects to be copied. This is likely to be true for a class like `Lock`, because it rarely makes sense to have "copies" of synchronization primitives. When copying makes no sense for an RAII class, you should prohibit it. Item 6 explains how to do that: declare the copying operations private. For `Lock`, that could look like this:

Click here to view code image

```
class Lock: private Uncopyable {               // prohibit
copying — see
public:                                        // Item 6

...                                           // as before

};
```

- **Reference-count the underlying resource.** Sometimes it's desirable to hold on to a resource until the last object using it has been destroyed. When that's the case, copying an RAII object should increment the count of the number of objects referring to the resource. This is the meaning of “copy” used by `tr1::shared_ptr`.

  Often, RAII classes can implement reference-counting copying behavior by containing a `tr1::shared_ptr` data member. For example, if `Lock` wanted to employ reference counting, it could change the type of `mutexPtr` from `Mutex*` to `tr1::shared_ptr<Mutex>`. Unfortunately, `tr1::shared_ptr`'s default behavior is to delete what it points to when the reference count goes to zero, and that's not what we want. When we're done with a `Mutex`, we want to unlock it, not delete it.

  Fortunately, `tr1::shared_ptr` allows specification of a “deleter” — a function or function object to be called when the reference count goes to zero. (This functionality does not exist for `auto_ptr`, which *always* deletes its pointer.) The deleter is an optional second parameter to the `tr1::shared_ptr` constructor, so the code would look like this:

Click here to view code image

```
class Lock {
public:
  explicit Lock(Mutex *pm)        // init shared_ptr with the
Mutex
  : mutexPtr(pm, unlock)          // to point to and the
unlock func
  {                               // as the deleter

    lock(mutexPtr.get());    // see Item 15 for info on "get"
  }
private:
  std::tr1::shared_ptr<Mutex> mutexPtr;     // use shared_ptr
};                                          // instead of raw
pointer
```

In this example, notice how the `Lock` class no longer declares a destructor. That's because there's no need to. Item 5 explains that a class's destructor (regardless of whether it is compiler-generated or user-defined) automatically invokes the destructors of the class's non-static data members. In this example, that's `mutexPtr`. But `mutexPtr`'s destructor will automatically call the `tr1::shared_ptr`'s deleter — `unlock`, in this case — when the mutex's reference count goes to zero. (People looking at the class's source code would probably appreciate a comment indicating that you didn't forget about destruction, you're just relying on the default compiler-generated behavior.)

- **Copy the underlying resource.** Sometimes you can have as many copies of a resource as you like, and the only reason you need a resource-managing class is to make sure that each copy is released when you're done with it. In that case, copying the resource-managing object should also copy the resource it wraps. That is, copying a resource-managing object performs a “deep copy.”

  Some implementations of the standard `string` type consist of pointers to heap memory, where the characters making up the string are stored. Objects of such `string`s contain a pointer to the heap memory. When a `string` object is copied, a copy is made of both the pointer and the memory it points to. Such `string`s exhibit deep copying.

- **Transfer ownership of the underlying resource.** On rare occasion, you may wish to make sure that only one RAII object refers to a raw resource and that when the RAII object is copied, ownership of the resource is transferred from the copied object to the copying object. As explained in Item 13, this is the meaning of “copy” used by `auto_ptr`.

The copying functions (copy constructor and copy assignment operator) may be generated by compilers, so unless the compiler-generated versions will do what you want (Item 5 explains the default behavior), you'll need to write them yourself. In some cases, you'll also want to support generalized versions of these functions. Such versions are described in Item 45.

Things to Remember

- Copying an RAII object entails copying the resource it manages, so the copying behavior of the resource determines the copying behavior of the RAII object.
- Common RAII class copying behaviors are disallowing copying and performing reference counting, but other behaviors are possible.

Item 15: Provide access to raw resources in resource-managing classes.

Resource-managing classes are wonderful. They're your bulwark against resource leaks, the absence of such leaks being a fundamental characteristic of well-designed systems. In a perfect world, you'd rely on such classes for all your interactions with resources, never sullying your hands with direct access to raw resources. But the world is not perfect. Many APIs refer to resources directly, so unless you plan to foreswear use of such APIs (something that's rarely practical), you'll have to bypass resource-managing objects and deal with raw resources from time to time.

For example, Item 13 introduces the idea of using smart pointers like `auto_ptr` or `tr1::shared_ptr` to hold the result of a call to a factory function like `createInvestment`:

```
std::tr1::shared_ptr<Investment>
pInv(createInvestment());  // from Item 13
```

Suppose that a function you'd like to use when working with `Investment` objects is this:

Click here to view code image

```
int daysHeld(const Investment *pi);        // return number
of days
                                           // investment has
been held
```

You'd like to call it like this,

```
int days = daysHeld(pInv);                // error!
```

but the code won't compile: `daysHeld` wants a raw `Investment*` pointer, but you're passing an object of type `tr1::shared_ptr<Investment>`.

You need a way to convert an object of the RAII class (in this case, `tr1::shared_ptr`) into the raw resource it contains (e.g., the underlying `Investment*`). There are two general ways to do it: explicit conversion and implicit conversion.

`tr1::shared_ptr` and `auto_ptr` both offer a `get` member function to perform an explicit conversion, i.e., to return (a copy of) the raw pointer inside the smart pointer object:

Click here to view code image

```
int days = daysHeld(pInv.get());            // fine, passes
the raw pointer
                                            // in pInv to
daysHeld
```

Like virtually all smart pointer classes, `tr1::shared_ptr` and `auto_ptr` also overload the pointer dereferencing operators (`operator->` and `operator*`), and this allows implicit conversion to the underlying raw pointers:

Click here to view code image

```
class Investment {                         // root class for
a hierarchy
public:                                    // of investment
types
  bool isTaxFree() const;
  ...
};
Investment* createInvestment();                    // factory
function

std::tr1::shared_ptr<Investment>                   // have
tr1::shared_ptr
  pi1(createInvestment());                         // manage
a resource

bool taxable1 = !(pi1->isTaxFree());               // access
resource
                                                   // via
operator->
```

```
...
std::auto_ptr<Investment> pi2(createInvestment()); // have
auto_ptr
                                                   // manage
a
                                                   //
resource

bool taxable2 = !((*pi2).isTaxFree());             // access
resource
                                                   // via
operator*
...
```

Because it is sometimes necessary to get at the raw resource inside an RAII object, some RAII class designers grease the skids by offering an implicit conversion function. For example, consider this RAII class for fonts that are native to a C API:

Click here to view code image

```
FontHandle getFont();               // from C API—params
omitted
                                    // for simplicity

void releaseFont(FontHandle fh);    // from the same C API

class Font {                           // RAII class
public:
  explicit Font(FontHandle fh)         // acquire resource;
  : f(fh)                              // use pass-by-value,
because the
  {}                                   // C API does

  ~Font() { releaseFont(f); }          // release resource
  ...                                  // handle copying (see
Item 14)
private:
  FontHandle f;                        // the raw font
resource
};
```

Assuming there's a large font-related C API that deals entirely with `FontHandle`s, there will be a frequent need to convert from `Font` objects to

`FontHandle`s. The `Font` class could offer an explicit conversion function such as `get`:

Click here to view code image

```
class Font {
public:
  ...
  FontHandle get() const { return f; }  // explicit
conversion function
  ...
};
```

Unfortunately, this would require that clients call `get` every time they want to communicate with the API:

Click here to view code image

```
void changeFontSize(FontHandle f, int newSize);     // from
the C API

Font f(getFont());
int newFontSize;

...

changeFontSize(f.get(), newFontSize);               //
explicitly convert
                                                    // Font
to FontHandle
```

Some programmers might find the need to explicitly request such conversions off-putting enough to avoid using the class. That, in turn, would increase the chances of leaking fonts, the very thing the `Font` class is designed to prevent.

The alternative is to have `Font` offer an implicit conversion function to its `FontHandle`:

Click here to view code image

```
class Font {
public:
  ...
  operator FontHandle() const        // implicit conversion
function
  { return f; }
  ...
};
```

That makes calling into the C API easy and natural:

[Click here to view code image](#)

```
Font f(getFont());
int newFontSize;

...

changeFontSize(f, newFontSize);     // implicitly convert
Font
                                    // to FontHandle
```

The downside is that implicit conversions increase the chance of errors. For example, a client might accidently create a `FontHandle` when a `Font` was intended:

[Click here to view code image](#)

```
Font f1(getFont());

...

FontHandle f2 = f1;                 // oops! meant to copy a
Font
                                    // object, but instead
implicitly
                                    // converted f1 into its
underlying
                                    // FontHandle, then
copied that
```

Now the program has a `FontHandle` being managed by the `Font` object `f1`, but the `FontHandle` is also available for direct use as `f2`. That's almost never good. For example, when `f1` is destroyed, the font will be released, and `f2` will dangle.

The decision about whether to offer explicit conversion from an RAII class to its underlying resource (e.g., via a `get` member function) or whether to allow implicit conversion is one that depends on the specific task the RAII class is designed to perform and the circumstances in which it is intended to be used. The best design is likely to be the one that adheres to Item 18's advice to make interfaces easy to use correctly and hard to use incorrectly. Often, an explicit conversion function like `get` is the preferable path, because it minimizes the chances of unintended type conversions. Sometime, however, the naturalness of use arising from implicit type conversions will tip the scales in that direction.

It may have occurred to you that functions returning the raw resource inside an RAII class are contrary to encapsulation. That's true, but it's not the design disaster it may at first appear. RAII classes don't exist to encapsulate something; they exist to ensure that a particular action—resource release—takes place. If desired, encapsulation of the resource can be layered on top of this primary functionality, but it's not necessary. Furthermore, some RAII classes combine true encapsulation of implementation with very loose encapsulation of the underlying resource. For example, `tr1::shared_ptr` encapsulates all its reference-counting machinery, but it still offers easy access to the raw pointer it contains. Like most well-designed classes, it hides what clients don't need to see, but it makes available those things that clients honestly need to access.

Things to Remember

- APIs often require access to raw resources, so each RAII class should offer a way to get at the resource it manages.
- Access may be via explicit conversion or implicit conversion. In general, explicit conversion is safer, but implicit conversion is more convenient for clients.

Item 16: Use the same form in corresponding uses of `new` and `delete`.

What's wrong with this picture?

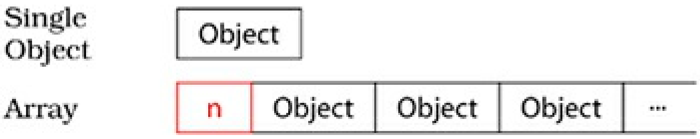
```
std::string *stringArray = new std::string[100];

...
```

```
delete stringArray;
```

Everything appears to be in order. The `new` is matched with a `delete`. Still, something is quite wrong. The program's behavior is undefined. At the very least, 99 of the 100 `string` objects pointed to by `stringArray` are unlikely to be properly destroyed, because their destructors will probably never be called.

When you employ a *`new` expression* (i.e., dynamic creation of an object via a use of `new`), two things happen. First, memory is allocated (via a function named `operator new`—see Items 49 and 51). Second, one or more constructors are called for that memory. When you employ a *`delete` expression* (i.e., use `delete`), two other things happen: one or more destructors are called for the memory, then the memory is deallocated (via a function named `operator delete`—see Item 51). The big question for `delete` is this: *how many* objects reside in the memory being deleted? The answer to that determines how many destructors must be called.

Actually, the question is simpler: does the pointer being deleted point to a single object or to an array of objects? It's a critical question, because the memory layout for single objects is generally different from the memory layout for arrays. In particular, the memory for an array usually includes the size of the array, thus making it easy for `delete` to know how many destructors to call. The memory for a single object lacks this information. You can think of the different layouts as looking like this, where `n` is the size of the array:



This is just an example, of course. Compilers aren't required to implement things this way, though many do.

When you use `delete` on a pointer, the only way for `delete` to know whether the array size information is there is for you to tell it. If you use brackets in your use of `delete`, `delete` assumes an array is pointed to. Otherwise, it assumes that a single object is pointed to:

Click here to view code image

```
std::string *stringPtr1 = new std::string;
```

```
std::string *stringPtr2 = new std::string[100];
...

delete stringPtr1;                      // delete an object

delete [] stringPtr2;                   // delete an array
of objects
```

What would happen if you used the “`[]`” form on `stringPtr1`? The result is undefined, but it's unlikely to be pretty. Assuming the layout above, `delete` would read some memory and interpret what it read as an array size, then start invoking that many destructors, oblivious to the fact that the memory it's working on not only isn't in the array, it's also probably not holding objects of the type it's busy destructing.

What would happen if you didn't use the “`[]`” form on `stringPtr2`? Well, that's undefined too, but you can see how it would lead to too few destructors being called. Furthermore, it's undefined (and sometimes harmful) for built-in types like `int`s, too, even though such types lack destructors.

The rule is simple: if you use `[]` in a `new` expression, you must use `[]` in the corresponding `delete` expression. If you don't use `[]` in a `new` expression, don't use `[]` in the matching `delete` expression.

This is a particularly important rule to bear in mind when you are writing a class containing a pointer to dynamically allocated memory and also offering multiple constructors, because then you must be careful to use the *same form* of `new` in all the constructors to initialize the pointer member. If you don't, how will you know what form of `delete` to use in your destructor?

This rule is also noteworthy for the `typedef`-inclined, because it means that a `typedef`'s author must document which form of `delete` should be employed when `new` is used to conjure up objects of the `typedef` type. For example, consider this `typedef`:

Click here to view code image

```
typedef std::string AddressLines[4];   // a person's address
has 4 lines,
                                       // each of which is a
string
```

Because `AddressLines` is an array, this use of `new`,

Click here to view code image

```
std::string *pal = new AddressLines;    // note that "new
AddressLines"
                                        // returns a string*,
just like
                                        // "new string[4]"
would
```

must be matched with the *array* form of `delete`:

```
delete pal;                            // undefined!

delete [] pal;                         // fine
```

To avoid such confusion, abstain from `typedef`s for array types. That's easy, because the standard C++ library (see Item 54) includes `string` and `vector`, and those templates reduce the need for dynamically allocated arrays to nearly zero. Here, for example, `AddressLines` could be defined to be a `vector` of `string`s, i.e., the type `vector<string>`.

Things to Remember

- If you use `[]` in a `new` expression, you must use `[]` in the corresponding `delete` expression. If you don't use `[]` in a `new` expression, you mustn't use `[]` in the corresponding `delete` expression.

Item 17: Store `new`ed objects in smart pointers in standalone statements.

Suppose we have a function to reveal our processing priority and a second function to do some processing on a dynamically allocated `Widget` in accord with a priority:

```
int priority();
void processWidget(std::tr1::shared_ptr<Widget> pw, int
priority);
```

Mindful of the wisdom of using objects to manage resources (see Item 13), `processWidget` uses a smart pointer (here, a `tr1::shared_ptr`) for the dynamically allocated `Widget` it processes.

Consider now a call to `processWidget`:

```
processWidget(new Widget, priority());
```

Wait, don't consider that call. It won't compile. `tr1::shared_ptr`'s constructor taking a raw pointer is `explicit`, so there's no implicit conversion from the raw pointer returned by the expression "`new Widget`" to the `tr1::shared_ptr` required by `processWidget`. The following code, however, will compile:

```
processWidget(std::tr1::shared_ptr<Widget>(new Widget),
priority());
```

Surprisingly, although we're using object-managing resources everywhere here, this call may leak resources. It's illuminating to see how.

Before compilers can generate a call to `processWidget`, they have to evaluate the arguments being passed as its parameters. The second argument is just a call to the function `priority`, but the first argument, ("`std::tr1::shared_ptr<Widget>(new Widget)`") consists of two parts:

- Execution of the expression "`new Widget`".
- A call to the `tr1::shared_ptr` constructor.

Before `processWidget` can be called, then, compilers must generate code to do these three things:

- Call `priority`.
- Execute "`new Widget`".
- Call the `tr1::shared_ptr` constructor.

C++ compilers are granted considerable latitude in determining the order in which these things are to be done. (This is different from the way languages like Java and C# work, where function parameters are always evaluated in a particular order.) The "`new Widget`" expression must be executed before the `tr1::shared_ptr` constructor can be called, because the result of the expression is passed as an argument to the `tr1::shared_ptr` constructor, but the call to `priority` can be performed first, second, or third. If compilers choose to perform it second (something that may allow them to generate more efficient code), we end up with this sequence of operations:

1. Execute “`new Widget`”.
2. Call `priority`.
3. Call the `tr1::shared_ptr` constructor.

But consider what will happen if the call to `priority` yields an exception. In that case, the pointer returned from “`new Widget`” will be lost, because it won't have been stored in the `tr1::shared_ptr` we were expecting would guard against resource leaks. A leak in the call to `processWidget` can arise because an exception can intervene between the time a resource is created (via “`new Widget`”) and the time that resource is turned over to a resource-managing object.

The way to avoid problems like this is simple: use a separate statement to create the `Widget` and store it in a smart pointer, then pass the smart pointer to `processWidget`:

Click here to view code image

```
std::tr1::shared_ptr<Widget> pw(new Widget);  // store newed
object
                                              // in a smart
pointer in a
                                              // standalone
statement

processWidget(pw, priority());                // this call
won't leak
```

This works because compilers are given less leeway in reordering operations *across* statements than *within* them. In this revised code, the “`new Widget`” expression and the call to the `tr1::shared_ptr` constructor are in a different statement from the one calling `priority`, so compilers are not allowed to move the call to `priority` between them.

Things to Remember

- Store `new`ed objects in smart pointers in standalone statements. Failure to do this can lead to subtle resource leaks when exceptions are thrown.

4. Designs and Declarations

Software designs — approaches to getting the software to do what you want it to do — typically begin as fairly general ideas, but they eventually become detailed enough to allow for the development of specific interfaces. These interfaces must then be translated into C++ declarations. In this chapter, we attack the problem of designing and declaring good C++ interfaces. We begin with perhaps the most important guideline about designing interfaces of any kind: that they should be easy to use correctly and hard to use incorrectly. That sets the stage for a number of more specific guidelines addressing a wide range of topics, including correctness, efficiency, encapsulation, maintainability, extensibility, and conformance to convention.

The material that follows isn't everything you need to know about good interface design, but it highlights some of the most important considerations, warns about some of the most frequent errors, and provides solutions to problems often encountered by class, function, and template designers.

Item 18: Make interfaces easy to use correctly and hard to use incorrectly

C++ is awash in interfaces. Function interfaces. Class interfaces. Template interfaces. Each interface is a means by which clients interact with your code. Assuming you're dealing with reasonable people, those clients are trying to do a good job. They *want* to use your interfaces correctly. That being the case, if they use one incorrectly, your interface is at least partially to blame. Ideally, if an attempted use of an interface won't do what the client expects, the code won't compile; and if the code does compile, it will do what the client wants.

Developing interfaces that are easy to use correctly and hard to use incorrectly requires that you consider the kinds of mistakes that clients might make. For example, suppose you're designing the constructor for a class representing dates in time:

```
class Date {
public:
  Date(int month, int day, int year);
  ...
};
```

At first glance, this interface may seem reasonable (at least in the USA), but there are at least two errors that clients might easily make. First, they might pass parameters in the wrong order:

```
Date d(30, 3, 1995);              // Oops! Should be "3, 30"
, not "30, 3"
```

Second, they might pass an invalid month or day number:

```
Date d(3, 40, 1995);              // Oops! Should be "3, 30"
, not "3, 40"
```

(This last example may look silly, but remember that on a keyboard, 4 is next to 3. Such “off by one” typing errors are not uncommon.)

Many client errors can be prevented by the introduction of new types. Indeed, the type system is your primary ally in preventing undesirable code from compiling. In this case, we can introduce simple wrapper types to distinguish days, months, and years, then use these types in the `Date` constructor:

Click here to view code image

```
struct Day {            struct Month {                struct
Year {
  explicit Day(int d)     explicit Month(int
m)         explicit Year(int y)
  :val(d) {}              :val(m)
{}                    :val(y){}

  int val;                int val;                      int
val;
};                      };                            };
class Date {
public:
Date(const Month& m, const Day& d, const Year& y);
...
};
Date d(30, 3, 1995);                      // error! wrong
types
```

```
Date d(Day(30), Month(3), Year(1995));    // error! wrong
types

Date d(Month(3), Day(30), Year(1995));    // okay, types are
correct
```

Making `Day`, `Month`, and `Year` full-fledged classes with encapsulated data would be better than the simple use of structs above (see Item 22), but even structs suffice to demonstrate that the judicious introduction of new types can work wonders for the prevention of interface usage errors.

Once the right types are in place, it can sometimes be reasonable to restrict the values of those types. For example, there are only 12 valid month values, so the `Month` type should reflect that. One way to do this would be to use an enum to represent the month, but enums are not as type-safe as we might like. For example, enums can be used like `int`s (see Item 2). A safer solution is to predefine the set of all valid `Month`s:

Click here to view code image

```
class Month {
public:
  static Month Jan() { return Month(1); }   // functions
returning all valid
  static Month Feb() { return Month(2); }   // Month values;
see below for
  ...                                       // why these are
functions, not
  static Month Dec() { return Month(12); }  // objects

  ...                                       // other member
functions

private:
  explicit Month(int m);                    // prevent
creation of new
                                            // Month values

  ...                                       // month-specific
data
};
Date d(Month::Mar(), Day(30), Year(1995));
```

If the idea of using functions instead of objects to represent specific months strikes you as odd, it may be because you have forgotten that reliable initialization of non-local static objects can be problematic. Item 4 can refresh your memory.

Another way to prevent likely client errors is to restrict what can be done with a type. A common way to impose restrictions is to add `const`. For example, Item 3 explains how `const`-qualifying the return type from `operator*` can prevent clients from making this error for user-defined types:

```
if (a * b = c) ...                    // oops, meant to do a
comparison!
```

In fact, this is just a manifestation of another general guideline for making types easy to use correctly and hard to use incorrectly: unless there's a good reason not to, have your types behave consistently with the built-in types. Clients already know how types like `int` behave, so you should strive to have your types behave the same way whenever reasonable. For example, assignment to `a*b` isn't legal if `a` and `b` are `int`s, so unless there's a good reason to diverge from this behavior, it should be illegal for your types, too. When in doubt, do as the `int`s do.

The real reason for avoiding gratuitous incompatibilities with the built-in types is to offer interfaces that behave consistently. Few characteristics lead to interfaces that are easy to use correctly as much as consistency, and few characteristics lead to aggravating interfaces as much as inconsistency. The interfaces to STL containers are largely (though not perfectly) consistent, and this helps make them fairly easy to use. For example, every STL container has a member function named `size` that tells how many objects are in the container. Contrast this with Java, where you use the `length` *property* for arrays, the `length` *method* for `String`s, and the `size` *method* for `List`s; and with .NET, where `Array`s have a property named `Length`, while `ArrayList`s have a property named `Count`. Some developers think that integrated development environments (IDEs) render such inconsistencies unimportant, but they are mistaken. Inconsistency imposes mental friction into a developer's work that no IDE can fully remove.

Any interface that requires that clients remember to do something is prone to incorrect use, because clients can forget to do it. For example, Item 13 introduces a factory function that returns pointers to dynamically allocated objects in an `Investment` hierarchy:

[Click here to view code image](#)

```
Investment* createInvestment();    // from Item 13; parameters
omitted
                                   // for simplicity
```

To avoid resource leaks, the pointers returned from `createInvestment` must eventually be deleted, but that creates an opportunity for at least two types of client errors: failure to delete a pointer, and deletion of the same pointer more than once.

Item 13 shows how clients can store `createInvestment`'s return value in a smart pointer like `auto_ptr` or `tr1::shared_ptr`, thus turning over to the smart pointer the responsibility for using `delete`. But what if clients forget to use the smart pointer? In many cases, a better interface decision would be to preempt the problem by having the factory function return a smart pointer in the first place:

```
std::tr1::shared_ptr<Investment> createInvestment();
```

This essentially forces clients to store the return value in a `tr1::shared_ptr`, all but eliminating the possibility of forgetting to delete the underlying `Investment` object when it's no longer being used.

In fact, returning a `tr1::shared_ptr` makes it possible for an interface designer to prevent a host of other client errors regarding resource release, because, as Item 14 explains, `tr1::shared_ptr` allows a resource-release function — a "deleter" — to be bound to the smart pointer when the smart pointer is created. (`auto_ptr` has no such capability.)

Suppose clients who get an `Investment*` pointer from `createInvestment` are expected to pass that pointer to a function called `getRidOfInvestment` instead of using `delete` on it. Such an interface would open the door to a new kind of client error, one where clients use the wrong resource-destruction mechanism (i.e., `delete` instead of `getRidOfInvestment`). The implementer of `createInvestment` can forestall such problems by returning a `tr1::shared_ptr` with `getRidOfInvestment` bound to it as its deleter.

`tr1::shared_ptr` offers a constructor taking two arguments: the pointer to be managed and the deleter to be called when the reference count goes to zero. This suggests that the way to create a null `tr1::shared_ptr` with `getRidOfInvestment` as its deleter is this:

Click here to view code image

```
std::tr1::shared_ptr<Investment>      // attempt to create a
null
  pInv(0, getRidOfInvestment);        // shared_ptr with a
custom deleter;
                                      // this won't compile
```

Alas, this isn't valid C++. The `tr1::shared_ptr` constructor insists on its first parameter being a *pointer*, and 0 isn't a pointer, it's an `int`. Yes, it's *convertible* to a pointer, but that's not good enough in this case; `tr1::shared_ptr` insists on an actual pointer. A cast solves the problem:

Click here to view code image

```
std::tr1::shared_ptr<Investment>      // create a null
shared_ptr with
  pInv(static_cast<Investment*>(0),   // getRidOfInvestment
as its
        getRidOfInvestment);          // deleter; see Item 27
for info on
                                      // static_cast
```

This means that the code for implementing `createInvestment` to return a `tr1::shared_ptr` with `getRidOfInvestment` as its deleter would look something like this:

Click here to view code image

```
std::tr1::shared_ptr<Investment> createInvestment()
{
  std::tr1::shared_ptr<Investment>
retVal(static_cast<Investment*>(0),
                                          getRidOfInvestment);

  retVal = ... ;                         // make retVal point
to the
                                         // correct object

  return retVal;
}
```

Of course, if the raw pointer to be managed by `retVal` could be determined prior to creating `retVal`, it would be better to pass the raw pointer to `retVal`'s constructor instead of initializing `retVal` to null and then making an assignment to it. For details on why, consult Item 26.

An especially nice feature of `tr1::shared_ptr` is that it automatically uses its per-pointer deleter to eliminate another potential client error, the "cross-DLL problem." This problem crops up when an object is created using `new` in one dynamically linked library (DLL) but is `delete`d in a different DLL. On many platforms, such cross-DLL `new`/`delete` pairs lead to runtime errors. `tr1::shared_ptr` avoids the problem, because its default deleter uses `delete` from the same DLL where the `tr1::shared_ptr` is created. This means, for example, that if `Stock` is a class derived from `Investment` and `createInvestment` is implemented like this,

Click here to view code image

```
std::tr1::shared_ptr<Investment> createInvestment()
{
  return std::tr1::shared_ptr<Investment>(new Stock);
}
```

the returned `tr1::shared_ptr` can be passed among DLLs without concern for the cross-DLL problem. The `tr1::shared_ptr`s pointing to the `Stock` keep track of which DLL's `delete` should be used when the reference count for the `Stock` becomes zero.

This Item isn't about `tr1::shared_ptr` — it's about making interfaces easy to use correctly and hard to use incorrectly — but `tr1::shared_ptr` is such an easy way to eliminate some client errors, it's worth an overview of the cost of using it. The most common implementation of `tr1::shared_ptr` comes from Boost (see Item 55). Boost's `shared_ptr` is twice the size of a raw pointer, uses dynamically allocated memory for bookkeeping and deleter-specific data, uses a virtual function call when invoking its deleter, and incurs thread synchronization overhead when modifying the reference count in an application it believes is multithreaded. (You can disable multithreading support by defining a preprocessor symbol.) In short, it's bigger than a raw pointer, slower than a raw pointer, and uses auxiliary dynamic memory. In many applications, these additional runtime costs will be unnoticeable, but the reduction in client errors will be apparent to everyone.

Things to Remember

- Good interfaces are easy to use correctly and hard to use incorrectly. You should strive for these characteristics in all your interfaces.
- Ways to facilitate correct use include consistency in interfaces and behavioral compatibility with built-in types.
- Ways to prevent errors include creating new types, restricting operations on types, constraining object values, and eliminating client resource management responsibilities.
- `tr1::shared_ptr` supports custom deleters. This prevents the cross-DLL problem, can be used to automatically unlock mutexes (see Item 14), etc.

Item 19: Treat class design as type design

In C++, as in other object-oriented programming languages, defining a new class defines a new type. Much of your time as a C++ developer will thus be spent augmenting your type system. This means you're not just a class designer, you're a *type* designer. Overloading functions and operators, controlling memory allocation and deallocation, defining object initialization and finalization — it's all in your hands. You should therefore approach class design with the same care that language designers lavish on the design of the language's built-in types.

Designing good classes is challenging because designing good types is challenging. Good types have a natural syntax, intuitive semantics, and one or more efficient implementations. In C++, a poorly planned class definition can make it impossible to achieve any of these goals. Even the performance characteristics of a class's member functions may be affected by how they are declared.

How, then, do you design effective classes? First, you must understand the issues you face. Virtually every class requires that you confront the following questions, the answers to which often lead to constraints on your design:

- **How should objects of your new type be created and destroyed?** How this is done influences the design of your class's constructors and destructor, as well as its memory allocation and deallocation functions (`operator new`, `operator new[]`, `operator delete`, and `operator delete[]` — see Chapter 8), if you write them.

- **How should object initialization differ from object assignment?** The answer to this question determines the behavior of and the differences between your constructors and your assignment operators. It's important not to confuse initialization with assignment, because they correspond to different function calls (see Item 4).
- **What does it mean for objects of your new type to be passed by value?** Remember, the copy constructor defines how pass-by-value is implemented for a type.
- **What are the restrictions on legal values for your new type?** Usually, only some combinations of values for a class's data members are valid. Those combinations determine the invariants your class will have to maintain. The invariants determine the error checking you'll have to do inside your member functions, especially your constructors, assignment operators, and “setter” functions. It may also affect the exceptions your functions throw and, on the off chance you use them, your functions' exception specifications.
- **Does your new type fit into an inheritance graph?** If you inherit from existing classes, you are constrained by the design of those classes, particularly by whether their functions are virtual or non-virtual (see Items 34 and 36). If you wish to allow other classes to inherit from your class, that affects whether the functions you declare are virtual, especially your destructor (see Item 7).
- **What kind of type conversions are allowed for your new type?** Your type exists in a sea of other types, so should there be conversions between your type and other types? If you wish to allow objects of type `T1` to be *implicitly* converted into objects of type `T2`, you will want to write either a type conversion function in class `T1` (e.g., `operator T2`) or a non-`explicit` constructor in class `T2` that can be called with a single argument. If you wish to allow *explicit* conversions only, you'll want to write functions to perform the conversions, but you'll need to avoid making them type conversion operators or non-`explicit` constructors that can be called with one argument. (For an example of both implicit and explicit conversion functions, see Item 15.)
- **What operators and functions make sense for the new type?** The

answer to this question determines which functions you'll declare for your class. Some functions will be member functions, but some will not (see Items 23, 24, and 46).

- **What standard functions should be disallowed?** Those are the ones you'll need to declare `private` (see Item 6).
- **Who should have access to the members of your new type?** This question helps you determine which members are public, which are protected, and which are private. It also helps you determine which classes and/or functions should be friends, as well as whether it makes sense to nest one class inside another.
- **What is the "undeclared interface" of your new type?** What kind of guarantees does it offer with respect to performance, exception safety (see Item 29), and resource usage (e.g., locks and dynamic memory)? The guarantees you offer in these areas will impose constraints on your class implementation.
- **How general is your new type?** Perhaps you're not really defining a new type. Perhaps you're defining a whole *family* of types. If so, you don't want to define a new class, you want to define a new class *template*.
- **Is a new type really what you need?** If you're defining a new derived class only so you can add functionality to an existing class, perhaps you'd better achieve your goals by simply defining one or more non-member functions or templates.

These questions are difficult to answer, so defining effective classes can be challenging. Done well, however, user-defined classes in C++ yield types that are at least as good as the built-in types, and that makes all the effort worthwhile.

Things to Remember

- Class design is type design. Before defining a new type, be sure to consider all the issues discussed in this Item.

Item 20: Prefer pass-by-reference-to-`const` to pass-by-value

By default, C++ passes objects to and from functions by value (a characteristic it inherits from C). Unless you specify otherwise, function parameters are initialized with *copies* of the actual arguments, and function callers get back a *copy* of the value returned by the function. These copies are produced by the objects' copy constructors. This can make pass-by-value an expensive operation. For example, consider the following class hierarchy:

Click here to view code image

```
class Person {
public:
  Person();                            // parameters omitted
for simplicity
  virtual ~Person();                   // see Item 7 for why
this is virtual
  ...

private:
  std::string name;
  std::string address;
};
class Student: public Person {
public:
  Student();                           // parameters again
omitted virtua
  ~Student();
  ...

private:
  std::string schoolName;
  std::string schoolAddress;
};
```

Now consider the following code, in which we call a function, `validateStudent`, that takes a `Student` argument (by value) and returns whether it has been validated:

Click here to view code image

```
bool validateStudent(Student s);           // function taking
a Student
                                           // by value

Student plato;                             // Plato studied
under Socrates
```

```
bool platoIsOK = validateStudent(plato);    // call the
function
```

What happens when this function is called?

Clearly, the `Student` copy constructor is called to initialize the parameter `s` from `plato`. Equally clearly, `s` is destroyed when `validateStudent` returns. So the parameter-passing cost of this function is one call to the `Student` copy constructor and one call to the `Student` destructor.

But that's not the whole story. A `Student` object has two `string` objects within it, so every time you construct a `Student` object you must also construct two `string` objects. A `Student` object also inherits from a `Person` object, so every time you construct a `Student` object you must also construct a `Person` object. A `Person` object has two additional `string` objects inside it, so each `Person` construction also entails two more `string` constructions. The end result is that passing a `Student` object by value leads to one call to the `Student` copy constructor, one call to the `Person` copy constructor, and four calls to the `string` copy constructor. When the copy of the `Student` object is destroyed, each constructor call is matched by a destructor call, so the overall cost of passing a `Student` by value is six constructors and six destructors!

Now, this is correct and desirable behavior. After all, you *want* all your objects to be reliably initialized and destroyed. Still, it would be nice if there were a way to bypass all those constructions and destructions. There is: pass by reference-to-`const`:

```
bool validateStudent(const Student& s);
```

This is much more efficient: no constructors or destructors are called, because no new objects are being created. The `const` in the revised parameter declaration is important. The original version of `validateStudent` took a `Student` parameter by value, so callers knew that they were shielded from any changes the function might make to the `Student` they passed in; `validateStudent` would be able to modify only a *copy* of it. Now that the `Student` is being passed by reference, it's necessary to also declare it `const`, because otherwise callers would have to worry about `validateStudent` making changes to the `Student` they passed in.

Passing parameters by reference also avoids the *slicing problem*. When a derived class object is passed (by value) as a base class object, the base class copy constructor is called, and the specialized features that make the object behave like a derived class object are “sliced” off. You're left with a simple base class object — little surprise, since a base class constructor created it. This is almost never what you want. For example, suppose you're working on a set of classes for implementing a graphical window system:

Click here to view code image

```
class Window {
public:
  ...
  std::string name() const;           // return name of
window
  virtual void display() const;       // draw window and
contents
};

class Window-WithScrollBars: public Window {
public:
  ...
  virtual void display() const;
};
```

All `Window` objects have a name, which you can get at through the `name` function, and all windows can be displayed, which you can bring about by invoking the `display` function. The fact that `display` is virtual tells you that the way in which simple base class `Window` objects are displayed is apt to differ from the way in which the fancier `Window-WithScrollBars` objects are displayed (see Items 34 and 36).

Now suppose you'd like to write a function to print out a window's name and then display the window. Here's the *wrong* way to write such a function:

Click here to view code image

```
void printNameAndDisplay(Window w)         // incorrect!
parameter
{                                          // may be sliced!
  std::cout << w.name();
  w.display();
}
```

Consider what happens when you call this function with a `Window-WithScrollBars` object:

```
Window-WithScrollBars wwsb;

printNameAndDisplay(wwsb);
```

The parameter `w` will be constructed — it's passed by value, remember? — as a *`Window`* object, and all the specialized information that made `wwsb` act like a `Window-WithScrollBars` object will be sliced off. Inside `printNameAndDisplay`, `w` will always act like an object of class `Window` (because it *is* an object of class `Window`), regardless of the type of object passed to the function. In particular, the call to `display` inside `printNameAndDisplay` will *always* call `Window::display`, never `Window-WithScrollBars::display`.

The way around the slicing problem is to pass `w` by reference-to-`const`:

Click here to view code image

```
void printNameAndDisplay(const Window& w)    // fine,
parameter won't
{                                            // be sliced
  std::cout << w.name();
  w.display();
}
```

Now `w` will act like whatever kind of window is actually passed in.

If you peek under the hood of a C++ compiler, you'll find that references are typically implemented as pointers, so passing something by reference usually means really passing a pointer. As a result, if you have an object of a built-in type (e.g., an `int`), it's often more efficient to pass it by value than by reference. For built-in types, then, when you have a choice between pass-by-value and pass-by-reference-to-`const`, it's not unreasonable to choose pass-by-value. This same advice applies to iterators and function objects in the STL, because, by convention, they are designed to be passed by value. Implementers of iterators and function objects are responsible for seeing to it that they are efficient to copy and are not subject to the slicing problem. (This is an example of how the rules change, depending on the part of C++ you are using — see Item 1.)

Built-in types are small, so some people conclude that all small types are good candidates for pass-by-value, even if they're user-defined. This is shaky reasoning. Just because an object is small doesn't mean that calling its copy constructor is inexpensive. Many objects — most STL containers among them — contain little more than a pointer, but copying such objects entails copying everything they point to. That can be *very* expensive.

Even when small objects have inexpensive copy constructors, there can be performance issues. Some compilers treat built-in and user-defined types differently, even if they have the same underlying representation. For example, some compilers refuse to put objects consisting of only a `double` into a register, even though they happily place naked `double`s there on a regular basis. When that kind of thing happens, you can be better off passing such objects by reference, because compilers will certainly put pointers (the implementation of references) into registers.

Another reason why small user-defined types are not necessarily good pass-by-value candidates is that, being user-defined, their size is subject to change. A type that's small now may be bigger in a future release, because its internal implementation may change. Things can even change when you switch to a different C++ implementation. As I write this, for example, some implementations of the standard library's `string` type are *seven times* as big as others.

In general, the only types for which you can reasonably assume that pass-by-value is inexpensive are built-in types and STL iterator and function object types. For everything else, follow the advice of this Item and prefer pass-by-reference-to-`const` over pass-by-value.

Things to Remember

- Prefer pass-by-reference-to-`const` over pass-by-value. It's typically more efficient and it avoids the slicing problem.
- The rule doesn't apply to built-in types and STL iterator and function object types. For them, pass-by-value is usually appropriate.

Item 21: Don't try to return a reference when you must return an object

Once programmers grasp the efficiency implications of pass-by-value for objects (see Item 20), many become crusaders, determined to root out the evil of pass-by-value wherever it may hide. Unrelenting in their pursuit of pass-by-reference purity, they invariably make a fatal mistake: they start to pass references to objects that don't exist. This is not a good thing.

Consider a class for representing rational numbers, including a function for multiplying two rationals together:

Click here to view code image

```
class Rational {
public:
  Rational(int numerator = 0,               // see Item 24
for why this
           int denominator = 1);            // ctor isn't
declared explicit

  ...

private:
  int n, d;                                 // numerator and
denominator

friend
   const Rational                           // see Item 3 for
why the
     operator*(const Rational& lhs,         // return type is
const
               const Rational& rhs);
};
```

This version of `operator*` is returning its result object by value, and you'd be shirking your professional duties if you failed to worry about the cost of that object's construction and destruction. You don't want to pay for such an object if you don't have to. So the question is this: do you have to pay?

Well, you don't have to if you can return a reference instead. But remember that a reference is just a *name*, a name for some *existing* object. Whenever you see the declaration for a reference, you should immediately ask yourself what it is another name for, because it must be another name for *something*. In the case of `operator*`, if the function is to return a reference, it must return a reference to some `Rational` object that already exists and that contains the product of the two objects that are to be multiplied together.

There is certainly no reason to expect that such an object exists prior to the call to `operator*`. That is, if you have

```
Rational a(1, 2);                         // a = 1/2
Rational b(3, 5);                         // b = 3/5

Rational c = a * b;                       // c should be 3/10
```

it seems unreasonable to expect that there already happens to exist a rational number with the value three-tenths. No, if `operator*` is to return a reference to such a number, it must create that number object itself.

A function can create a new object in only two ways: on the stack or on the heap. Creation on the stack is accomplished by defining a local variable. Using that strategy, you might try to write `operator*` this way:

Click here to view code image

```
const Rational& operator*(const Rational& lhs,    // warning!
bad code!
                          const Rational& rhs)
{
  Rational result(lhs.n * rhs.n, lhs.d * rhs.d);
  return result;
}
```

You can reject this approach out of hand, because your goal was to avoid a constructor call, and `result` will have to be constructed just like any other object. A more serious problem is that this function returns a reference to `result`, but `result` is a local object, and local objects are destroyed when the function exits. This version of `operator*`, then, doesn't return a reference to a `Rational` – it returns a reference to an *ex*-`Rational`; a *former* `Rational`; the empty, stinking, rotting carcass of what *used* to be a `Rational` but is no longer, because it has been destroyed. Any caller so much as *glancing* at this function's return value would instantly enter the realm of undefined behavior. The fact is, any function returning a reference to a local object is broken. (The same is true for any function returning a pointer to a local object.)

Let us consider, then, the possibility of constructing an object on the heap and returning a reference to it. Heap-based objects come into being through the use of `new`, so you might write a heap-based `operator*` like this:

Click here to view code image

```
const Rational& operator*(const Rational& lhs,    // warning!
more bad
                          const Rational& rhs)    // code!
{
  Rational *result = new Rational(lhs.n * rhs.n, lhs.d *
rhs.d);
  return *result;
}
```

Well, you *still* have to pay for a constructor call, because the memory allocated by `new` is initialized by calling an appropriate constructor, but now you have a different problem: who will apply `delete` to the object conjured up by your use of `new`?

Even if callers are conscientious and well intentioned, there's not much they can do to prevent leaks in reasonable usage scenarios like this:

Click here to view code image

```
Rational w, x, y, z;

w = x * y * z;                     // same as operator*
(operator*(x, y), z)
```

Here, there are two calls to `operator*` in the same statement, hence two uses of `new` that need to be undone with uses of `delete`. Yet there is no reasonable way for clients of `operator*` to make those calls, because there's no reasonable way for them to get at the pointers hidden behind the references being returned from the calls to `operator*`. This is a guaranteed resource leak.

But perhaps you notice that both the on-the-stack and on-the-heap approaches suffer from having to call a constructor for each result returned from `operator*`. Perhaps you recall that our initial goal was to avoid such constructor invocations. Perhaps you think you know a way to avoid all but one constructor call. Perhaps the following implementation occurs to you, an implementation based on `operator*` returning a reference to a *static* `Rational` object, one defined *inside* the function:

Click here to view code image

```
const Rational& operator*(const Rational& lhs,     // warning!
yet more
                          const Rational& rhs)    // bad
code!
{
  static Rational result;              // static object to
which a
                                       // reference will be
returned

  result = ... ;                       // multiply lhs by rhs
and put the
                                       // product inside
result
  return result;
}
```

Like all designs employing the use of static objects, this one immediately raises our thread-safety hackles, but that's its more obvious weakness. To see its deeper flaw, consider this perfectly reasonable client code:

Click here to view code image

```
bool operator==(const Rational& lhs,            // an
operator==
                const Rational& rhs);           // for
Rationals

Rational a, b, c, d;

...
if ((a * b) == (c * d))  {
    do whatever's appropriate when the products are equal;
} else    {
   do whatever's appropriate when they're not;
}
```

Guess what? The expression `((a*b) == (c*d))` will *always* evaluate to `true`, regardless of the values of `a`, `b`, `c`, and `d`!

This revelation is easiest to understand when the code is rewritten in its equivalent functional form:

```
if (operator==(operator*(a, b), operator*(c, d)))
```

Notice that when `operator==` is called, there will already be *two* active calls to `operator*`, each of which will return a reference to the static `Rational` object inside `operator*`. Thus, `operator==` will be asked to compare the value of the static `Rational` object inside `operator*` with the value of the static `Rational` object inside `operator*`. It would be surprising indeed if they did not compare equal. Always.

This should be enough to convince you that returning a reference from a function like `operator*` is a waste of time, but some of you are now thinking, “Well, if *one* static isn't enough, maybe a static *array* will do the trick....”

I can't bring myself to dignify this design with example code, but I can sketch why the notion should cause you to blush in shame. First, you must choose *n*, the size of the array. If *n* is too small, you may run out of places to store function return values, in which case you'll have gained nothing over the single-static design we just discredited. But if *n* is too big, you'll decrease the performance of your program, because *every* object in the array will be constructed the first time the function is called. That will cost you *n* constructors and *n* destructors[†], even if the function in question is called only once. If “optimization” is the process of improving software performance, this kind of thing should be called “pessimization.” Finally, think about how you'd put the values you need into the array's objects and what it would cost you to do it. The most direct way to move a value between objects is via assignment, but what is the cost of an assignment? For many types, it's about the same as a call to a destructor (to destroy the old value) plus a call to a constructor (to copy over the new value). But your goal is to avoid the costs of construction and destruction! Face it: this approach just isn't going to pan out. (No, using a `vector` instead of an array won't improve matters much.)

[†] The destructors will be called once at program shutdown.

The right way to write a function that must return a new object is to have that function return a new object. For `Rational`'s `operator*`, that means either the following code or something essentially equivalent:

Click here to view code image

```
inline const Rational operator*(const Rational& lhs, const
Rational& rhs)
{
  return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

Sure, you may incur the cost of constructing and destructing `operator*`'s return value, but in the long run, that's a small price to pay for correct behavior. Besides, the bill that so terrifies you may never arrive. Like all programming languages, C++ allows compiler implementers to apply optimizations to improve the performance of the generated code without changing its observable behavior, and it turns out that in some cases, construction and destruction of `operator*`'s return value can be safely eliminated. When compilers take advantage of that fact (and compilers often do), your program continues to behave the way it's supposed to, just faster than you expected.

It all boils down to this: when deciding between returning a reference and returning an object, your job is to make the choice that offers correct behavior. Let your compiler vendors wrestle with figuring out how to make that choice as inexpensive as possible.

Things to Remember

- Never return a pointer or reference to a local stack object, a reference to a heap-allocated object, or a pointer or reference to a local static object if there is a chance that more than one such object will be needed. (Item 4 provides an example of a design where returning a reference to a local static is reasonable, at least in single-threaded environments.)

Item 22: Declare data members `private`

Okay, here's the plan. First, we're going to see why data members shouldn't be public. Then we'll see that all the arguments against public data members apply equally to protected ones. That will lead to the conclusion that data members should be private, and at that point, we'll be done.

So, public data members. Why not?

Let's begin with syntactic consistency (see also Item 18). If data members aren't public, the only way for clients to access an object is via member functions. If everything in the public interface is a function, clients won't have to scratch their heads trying to remember whether to use parentheses when they want to access a member of the class. They'll just do it, because everything is a function. Over the course of a lifetime, that can save a lot of head scratching.

But maybe you don't find the consistency argument compelling. How about the fact that using functions gives you much more precise control over the accessibility of data members? If you make a data member public, everybody has read-write access to it, but if you use functions to get or set its value, you can implement no access, read-only access, and read-write access. Heck, you can even implement write-only access if you want to:

Click here to view code image

```
class AccessLevels {
public:
  ...
  int getReadOnly() const        { return readOnly; }

  void setReadWrite(int value)   { readWrite = value; }
  int getReadWrite() const       { return readWrite; }

  void setWriteOnly(int value)   { writeOnly = value; }

private:
  int noAccess;                         // no access to this
int

  int readOnly;                         // read-only access
to this int

  int readWrite;                        // read-write access
to this int

  int writeOnly;                        // write-only access
to this int
};
```

Such fine-grained access control is important, because many data members *should* be hidden. Rarely does every data member need a getter and setter.

Still not convinced? Then it's time to bring out the big gun: encapsulation. If you implement access to a data member through a function, you can later replace the data member with a computation, and nobody using your class will be any the wiser.

For example, suppose you are writing an application in which automated equipment is monitoring the speed of passing cars. As each car passes, its speed is computed and the value added to a collection of all the speed data collected so far:

Click here to view code image

```
class SpeedDataCollection {
  ...
public:
  void addValue(int speed);          // add a new data value

  double averageSoFar() const;       // return average speed

  ...
};
```

Now consider the implementation of the member function `averageSoFar`. One way to implement it is to have a data member in the class that is a running average of all the speed data so far collected. Whenever `averageSoFar` is called, it just returns the value of that data member. A different approach is to have `averageSoFar` compute its value anew each time it's called, something it could do by examining each data value in the collection.

The first approach (keeping a running average) makes each `SpeedDataCollection` object bigger, because you have to allocate space for the data members holding the running average, the accumulated total, and the number of data points. However, `averageSoFar` can be implemented very efficiently; it's just an inline function (see Item 30) that returns the value of the running average. Conversely, computing the average whenever it's requested will make `averageSoFar` run slower, but each `SpeedDataCollection` object will be smaller.

Who's to say which is best? On a machine where memory is tight (e.g., an embedded roadside device), and in an application where averages are needed only infrequently, computing the average each time is probably a better solution. In an application where averages are needed frequently, speed is of the essence, and memory is not an issue, keeping a running average will typically be preferable. The important point is that by accessing the average through a member function (i.e., by encapsulating it), you can interchange these different implementations (as well as any others you might think of), and clients will, at most, only have to recompile. (You can eliminate even that inconvenience by following the techniques described in Item 31.)

Hiding data members behind functional interfaces can offer all kinds of implementation flexibility. For example, it makes it easy to notify other objects when data members are read or written, to verify class invariants and function pre-and postconditions, to perform synchronization in threaded environments, etc. Programmers coming to C++ from languages like Delphi and C# will recognize such capabilities as the equivalent of "properties" in these other languages, albeit with the need to type an extra set of parentheses.

The point about encapsulation is more important than it might initially appear. If you hide your data members from your clients (i.e., encapsulate them), you can ensure that class invariants are always maintained, because only member functions can affect them. Furthermore, you reserve the right to change your implementation decisions later. If you don't hide such decisions, you'll soon find that even if you own the source code to a class, your ability to change anything public is extremely restricted, because too much client code will be broken. Public means unencapsulated, and practically speaking, unencapsulated means unchangeable, especially for classes that are widely used. Yet widely used classes are most in need of encapsulation, because they are the ones that can most benefit from the ability to replace one implementation with a better one.

The argument against protected data members is similar. In fact, it's identical, though it may not seem that way at first. The reasoning about syntactic consistency and fine-grained access control is clearly as applicable to protected data as to public, but what about encapsulation? Aren't protected data members more encapsulated than public ones? Practically speaking, the surprising answer is that they are not.

Item 23 explains that something's encapsulation is inversely proportional to the amount of code that might be broken if that something changes. The encapsulatedness of a data member, then, is inversely proportional to the amount of code that might be broken if that data member changes, e.g., if it's removed from the class (possibly in favor of a computation, as in `averageSoFar`, above).

Suppose we have a public data member, and we eliminate it. How much code might be broken? All the client code that uses it, which is generally an *unknowably large* amount. Public data members are thus completely unencapsulated. But suppose we have a protected data member, and we eliminate it. How much code might be broken now? All the derived classes that use it, which is, again, typically an *unknowably large* amount of code.

Protected data members are thus as unencapsulated as public ones, because in both cases, if the data members are changed, an unknowably large amount of client code is broken. This is unintuitive, but as experienced library implementers will tell you, it's still true. Once you've declared a data member public or protected and clients have started using it, it's very hard to change anything about that data member. Too much code has to be rewritten, retested, redocumented, or recompiled. From an encapsulation point of view, there are really only two access levels: `private` (which offers encapsulation) and everything else (which doesn't).

Things to Remember

- Declare data members `private`. It gives clients syntactically uniform access to data, affords fine-grained access control, allows invariants to be enforced, and offers class authors implementation flexibility.
- `protected` is no more encapsulated than `public`.

Item 23: Prefer non-member non-friend functions to member functions

Imagine a class for representing web browsers. Among the many functions such a class might offer are those to clear the cache of downloaded elements, clear the history of visited URLs, and remove all cookies from the system:

```
class WebBrowser {
public:
  ...
  void clearCache();
  void clearHistory();
  void removeCookies();
  ...
};
```

Many users will want to perform all these actions together, so `WebBrowser` might also offer a function to do just that:

Click here to view code image

```
class WebBrowser {
```

```
public:
  ...
  void clearEverything();                    // calls clearCache,
clearHistory,
                                             // and removeCookies
  ...
};
```

Of course, this functionality could also be provided by a non-member function that calls the appropriate member functions:

Click here to view code image

```
void clearBrowser(WebBrowser& wb)
{
  wb.clearCache();
  wb.clearHistory();
  wb.removeCookies();
}
```

So which is better, the member function `clearEverything` or the non-member function `clearBrowser`?

Object-oriented principles dictate that data and the functions that operate on them should be bundled together, and that suggests that the member function is the better choice. Unfortunately, this suggestion is incorrect. It's based on a misunderstanding of what being object-oriented means. Object-oriented principles dictate that data should be as *encapsulated* as possible. Counterintuitively, the member function `clearEverything` actually yields *less* encapsulation than the non-member `clearBrowser`. Furthermore, offering the non-member function allows for greater packaging flexibility for `WebBrowser`-related functionality, and that, in turn, yields fewer compilation dependencies and an increase in `WebBrowser` extensibility. The non-member approach is thus better than a member function in many ways. It's important to understand why.

We'll begin with encapsulation. If something is encapsulated, it's hidden from view. The more something is encapsulated, the fewer things can see it. The fewer things can see it, the greater flexibility we have to change it, because our changes directly affect only those things that can see what we change. The greater something is encapsulated, then, the greater our ability to change it. That's the reason we value encapsulation in the first place: it affords us the flexibility to change things in a way that affects only a limited number of clients.

Consider the data associated with an object. The less code that can see the data (i.e., access it), the more the data is encapsulated, and the more freely we can change characteristics of an object's data, such as the number of data members, their types, etc. As a coarse-grained measure of how much code can see a piece of data, we can count the number of functions that can access that data: the more functions that can access it, the less encapsulated the data.

Item 22 explains that data members should be private, because if they're not, an unlimited number of functions can access them. They have no encapsulation at all. For data members that *are* private, the number of functions that can access them is the number of member functions of the class plus the number of friend functions, because only members and friends have access to private members. Given a choice between a member function (which can access not only the private data of a class, but also private functions, enums, typedefs, etc.) and a non-member non-friend function (which can access none of these things) providing the same functionality, the choice yielding greater encapsulation is the non-member non-friend function, because it doesn't increase the number of functions that can access the private parts of the class. This explains why `clearBrowser` (the non-member non-friend function) is preferable to `clearEverything` (the member function): it yields greater encapsulation in the `WebBrowser` class.

At this point, two things are worth noting. First, this reasoning applies only to non-member *non-friend* functions. Friends have the same access to a class's private members that member functions have, hence the same impact on encapsulation. From an encapsulation point of view, the choice isn't between member and non-member functions, it's between member functions and non-member non-friend functions. (Encapsulation isn't the only point of view, of course. Item 24 explains that when it comes to implicit type conversions, the choice *is* between member and non-member functions.)

The second thing to note is that just because concerns about encapsulation dictate that a function be a non-member of one class doesn't mean it can't be a member of another class. This may prove a mild salve to programmers accustomed to languages where all functions *must* be in classes (e.g., Eiffel, Java, C#, etc.). For example, we could make `clearBrowser` a static member function of some utility class. As long as it's not part of (or a friend of)

`WebBrowser`, it doesn't affect the encapsulation of `WebBrowser`'s private members.

In C++, a more natural approach would be to make `clearBrowser` a non-member function in the same namespace as `WebBrowser`:

Click here to view code image

```
namespace WebBrowserStuff {

class WebBrowser { ... };

void clearBrowser(WebBrowser& wb);

  ...
}
```

This has more going for it than naturalness, however, because namespaces, unlike classes, can be spread across multiple source files. That's important, because functions like `clearBrowser` are *convenience functions*. Being neither members nor friends, they have no special access to `WebBrowser`, so they can't offer any functionality a `WebBrowser` client couldn't already get in some other way. For example, if `clearBrowser` didn't exist, clients could just call `clearCache`, `clearHistory`, and `removeCookies` themselves.

A class like `WebBrowser` might have a large number of convenience functions, some related to bookmarks, others related to printing, still others related to cookie management, etc. As a general rule, most clients will be interested in only some of these sets of convenience functions. There's no reason for a client interested only in bookmark-related convenience functions to be compilation dependent on, e.g., cookie-related convenience functions. The straightforward way to separate them is to declare bookmark-related convenience functions in one header file, cookie-related convenience functions in a different header file, printing-related convenience functions in a third, etc.:

Click here to view code image

```
// header "webbrowser.h" — header for class WebBrowser itself
// as well as "core" WebBrowser-related functionality
namespace WebBrowserStuff {

   class WebBrowser { ... };
```

```
     ...                                    // "core" related
functionality, e.g.
                                            // non-member
functions almost
                                            // all clients need
}
// header "webbrowserbookmarks.h"
namespace WebBrowserStuff {
  ...                                       // bookmark-related
convenience
}                                           // functions
// header "webbrowsercookies.h"
namespace WebBrowserStuff {
  ...                                       // cookie-related
convenience
}                                           // functions

...
```

Note that this is exactly how the standard C++ library is organized. Rather than having a single monolithic `<C++StandardLibrary>` header containing everything in the `std` namespace, there are dozens of headers (e.g., `<vector>`, `<algorithm>`, `<memory>`, etc.), each declaring *some* of the functionality in `std`. Clients who use only `vector`-related functionality aren't required to `#include <memory>`; clients who don't use `list` don't have to `#include <list>`. This allows clients to be compilation dependent only on the parts of the system they actually use. (See Item 31 for a discussion of other ways to reduce compilation dependencies.) Partitioning functionality in this way is not possible when it comes from a class's member functions, because a class must be defined in its entirety; it can't be split into pieces.

Putting all convenience functions in multiple header files — but one namespace — also means that clients can easily *extend* the set of convenience functions. All they have to do is add more non-member non-friend functions to the namespace. For example, if a `WebBrowser` client decides to write convenience functions related to downloading images, he or she just needs to create a new header file containing the declarations of those functions in the `WebBrowserStuff` namespace. The new functions are now as available and as integrated as all other convenience functions. This is another feature classes can't offer, because class definitions are closed to extension by clients. Sure, clients can derive new classes, but derived classes have no access to encapsulated (i.e., private) members in the base class, so such “extended functionality” has second-class status. Besides, as Item 7 explains, not all classes are designed to be base classes.

Things to Remember

- Prefer non-member non-friend functions to member functions. Doing so increases encapsulation, packaging flexibility, and functional extensibility.

Item 24: Declare non-member functions when type conversions should apply to all parameters

I noted in the Introduction to this book that having classes support implicit type conversions is generally a bad idea. Of course, there are exceptions to this rule, and one of the most common is when creating numerical types. For example, if you're designing a class to represent rational numbers, allowing implicit conversions from integers to rationals doesn't seem unreasonable. It's certainly no less reasonable than C++'s built-in conversion from `int` to `double` (and it's a lot more reasonable than C++'s built-in conversion from `double` to `int`). That being the case, you might start your `Rational` class this way:

Click here to view code image

```
class Rational {
public:
  Rational(int numerator = 0,        // ctor is deliberately
not explicit;
           int denominator = 1);     // allows implicit int-
to-Rational
                                     // conversions

  int numerator() const;             // accessors for
numerator and
  int denominator() const;           // denominator — see
Item 22

private:
  ...
};
```

You know you'd like to support arithmetic operations like addition, multiplication, etc., but you're unsure whether you should implement them via member functions, non-member functions, or, possibly, non-member functions that are friends. Your instincts tell you that when you're in doubt, you should be object-oriented. You know that, say, multiplication of rational numbers is related to the `Rational` class, so it seems natural to implement `operator*` for rational numbers inside the `Rational` class. Counterintuitively, Item 23 argues that the idea of putting functions inside the class they are associated with is sometimes *contrary* to object-oriented principles, but let's set that aside and investigate the idea of making `operator*` a member function of `Rational`:

Click here to view code image

```
class Rational {
public:
...

const Rational operator*(const Rational& rhs) const;
};
```

(If you're unsure why this function is declared the way it is — returning a `const` by-value result, but taking a reference-to-`const` as its argument — consult Items 3, 20, and 21.)

This design lets you multiply rationals with the greatest of ease:

Click here to view code image

```
Rational oneEighth(1, 8);
Rational oneHalf(1, 2);

Rational result = oneHalf * oneEighth;              // fine

result = result * oneEighth;                        // fine
```

But you're not satisfied. You'd also like to support mixed-mode operations, where `Rational`s can be multiplied with, for example, `int`s. After all, few things are as natural as multiplying two numbers together, even if they happen to be different types of numbers.

When you try to do mixed-mode arithmetic, however, you find that it works only half the time:

```
result = oneHalf * 2;                             // fine

result = 2 * oneHalf;                             // error!
```

This is a bad omen. Multiplication is supposed to be commutative, remember?

The source of the problem becomes apparent when you rewrite the last two examples in their equivalent functional form:

```
result = oneHalf.operator*(2);                    // fine

result = 2.operator*(oneHalf);                    // error!
```

The object `oneHalf` is an instance of a class that contains an `operator*`, so compilers call that function. However, the integer 2 has no associated class, hence no `operator*` member function. Compilers will also look for non-member `operator*`s (i.e., ones at namespace or global scope) that can be called like this:

```
result = operator*(2, oneHalf);                   // error!
```

But in this example, there is no non-member `operator*` taking an `int` and a `Rational`, so the search fails.

Look again at the call that succeeds. You'll see that its second parameter is the integer 2, yet `Rational::operator*` takes a `Rational` object as its argument. What's going on here? Why does 2 work in one position and not in the other?

What's going on is implicit type conversion. Compilers know you're passing an `int` and that the function requires a `Rational`, but they also know they can conjure up a suitable `Rational` by calling the `Rational` constructor with the `int` you provided, so that's what they do. That is, they treat the call as if it had been written more or less like this:

Click here to view code image

```
const Rational temp(2);              // create a temporary
                                     // Rational object from
2

result = oneHalf * temp;             // same as
```

```
oneHalf.operator*(temp);
```

Of course, compilers do this only because a non-`explicit` constructor is involved. If `Rational`'s constructor were `explicit`, neither of these statements would compile:

Click here to view code image

```
result = oneHalf * 2;                    // error! (with explicit
ctor);
                                         // can't convert 2 to
Rational

result = 2 * oneHalf;                    // same error, same
problem
```

That would fail to support mixed-mode arithmetic, but at least the behavior of the two statements would be consistent.

Your goal, however, is both consistency and support for mixed-mode arithmetic, i.e., a design where both of the above statements will compile. That brings us back to these two statements and why, even when `Rational`'s constructor is not `explicit`, one compiles and one does not:

Click here to view code image

```
result = oneHalf * 2;                    // fine (with non-
explicit ctor)

result = 2 * oneHalf;                    // error! (even with
non-explicit ctor)
```

It turns out that parameters are eligible for implicit type conversion *only if they are listed in the parameter list*. The implicit parameter corresponding to the object on which the member function is invoked — the one `this` points to — is *never* eligible for implicit conversions. That's why the first call compiles and the second one does not. The first case involves a parameter listed in the parameter list, but the second one doesn't.

You'd still like to support mixed-mode arithmetic, however, and the way to do it is by now perhaps clear: make `operator*` a non-member function, thus allowing compilers to perform implicit type conversions on *all* arguments:

Click here to view code image

```
class Rational {

  ...                                                  // contains
no operator*
};
const Rational operator*(const Rational& lhs,          // now a
non-member
                         const Rational& rhs)          // function
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}
Rational oneFourth(1, 4);
Rational result;

result = oneFourth * 2;                                // fine
result = 2 * oneFourth;                                // hooray,
it works!
```

This is certainly a happy ending to the tale, but there is a nagging worry. Should `operator*` be made a friend of the `Rational` class?

In this case, the answer is no, because `operator*` can be implemented entirely in terms of `Rational`'s public interface. The code above shows one way to do it. That leads to an important observation: the opposite of a member function is a *non-member* function, not a friend function. Too many C++ programmers assume that if a function is related to a class and should not be a member (due, for example, to a need for type conversions on all arguments), it should be a friend. This example demonstrates that such reasoning is flawed. Whenever you can avoid friend functions, you should, because, much as in real life, friends are often more trouble than they're worth. Sometimes friendship is warranted, of course, but the fact remains that just because a function shouldn't be a member doesn't automatically mean it should be a friend.

This Item contains the truth and nothing but the truth, but it's not the whole truth. When you cross the line from Object-Oriented C++ into Template C++ (see Item 1) and make `Rational` a class *template* instead of a class, there are new issues to consider, new ways to resolve them, and some surprising design implications. Such issues, resolutions, and implications are the topic of Item 46.

Things to Remember

- If you need type conversions on all parameters to a function (including the one that would otherwise be pointed to by the `this` pointer), the function must be a non-member.

Item 25: Consider support for a non-throwing `swap`

`swap` is an interesting function. Originally introduced as part of the STL, it's since become a mainstay of exception-safe programming (see Item 29) and a common mechanism for coping with the possibility of assignment to self (see Item 11). Because `swap` is so useful, it's important to implement it properly, but along with its singular importance comes a set of singular complications. In this Item, we explore what they are and how to deal with them.

To *swap* the values of two objects is to give each the other's value. By default, swapping is accomplished via the standard `swap` algorithm. Its typical implementation is exactly what you'd expect:

Click here to view code image

```
namespace std {

  template<typename T>           // typical implementation of
std::swap;
  void swap(T& a, T& b)          // swaps a's and b's values
  {
    T temp(a);
    a = b;
    b = temp;
  }
}
```

As long as your types support copying (via copy constructor and copy assignment operator), the default `swap` implementation will let objects of your types be swapped without your having to do any special work to support it.

However, the default `swap` implementation may not thrill you. It involves copying three objects: `a` to `temp`, `b` to `a`, and `temp` to `b`. For some types, none of these copies are really necessary. For such types, the default `swap` puts you on the fast track to the slow lane.

Foremost among such types are those consisting primarily of a pointer to another type that contains the real data. A common manifestation of this design approach is the “pimpl idiom” (“pointer to implementation” — see Item 31). A `Widget` class employing such a design might look like this:

Click here to view code image

```
class WidgetImpl {                            // class for
Widget data;
public:                                       // details are
unimportant
  ...

private:
  int a, b, c;                                // possibly lots
of data —
  std::vector<double> v;                      // expensive to
copy!
  ...
};

class Widget {                                // class using
the pimpl idiom
public:
  Widget(const Widget& rhs);

  Widget& operator=(const Widget& rhs)        // to copy a
Widget, copy its
  {                                           // WidgetImpl
object. For
   ...                                        // details on
implementing
   *pImpl = *(rhs.pImpl);                      // operator= in
general,
   ...                                         // see Items 10,
11, and 12.
  }
  ...

private:
  WidgetImpl *pImpl;                           // ptr to object
with this
};                                             // Widget's data
```

To swap the value of two `Widget` objects, all we really need to do is swap their `pImpl` pointers, but the default `swap` algorithm has no way to know that. Instead, it would copy not only three `Widget`s, but also three `WidgetImpl` objects. Very inefficient. Not a thrill.

What we'd like to do is tell `std::swap` that when `Widget`s are being swapped, the way to perform the swap is to `swap` their internal `pImpl` pointers. There is a way to say exactly that: specialize `std::swap` for `Widget`. Here's the basic idea, though it won't compile in this form:

Click here to view code image

```
namespace std {

  template<>                            // this is a
specialized version
  void swap<Widget>(Widget& a,          // of std::swap for
when T is
                    Widget& b)          // Widget
  {
    swap(a.pImpl, b.pImpl);             // to swap Widgets,
swap their
  }                                     // pImpl pointers;
this won't
compile
}
```

The “`template<>`” at the beginning of this function says that this is a *total template specialization* for `std::swap`, and the “`<Widget>`” after the name of the function says that the specialization is for when `T` is `Widget`. In other words, when the general `swap` template is applied to `Widget`s, this is the implementation that should be used. In general, we're not permitted to alter the contents of the `std` namespace, but we are allowed to totally specialize standard templates (like `swap`) for types of our own creation (such as `Widget`). That's what we're doing here.

As I said, though, this function won't compile. That's because it's trying to access the `pImpl` pointers inside `a` and `b`, and they're private. We could declare our specialization a friend, but the convention is different: it's to have `Widget` declare a public member function called `swap` that does the actual swapping, then specialize `std::swap` to call the member function:

Click here to view code image

```
class Widget {                     // same as above, except
for the
public:                            // addition of the swap
mem func
  ...
  void swap(Widget& other)
  {
    using std::swap;               // the need for this
declaration
                                   // is explained later in
this Item

    swap(pImpl, other.pImpl);      // to swap Widgets, swap
their
  }                                // pImpl pointers
  ...
};

namespace std {

  template<>                       // revised specialization
of
  void swap<Widget>(Widget& a,     // std::swap
                    Widget& b)
  {
    a.swap(b);                     // to swap Widgets, call
their
  }                                // swap member function
}
```

Not only does this compile, it's also consistent with the STL containers, all of which provide both public `swap` member functions and versions of `std::swap` that call these member functions.

Suppose, however, that `Widget` and `WidgetImpl` were class *templates* instead of classes, possibly so we could parameterize the type of the data stored in `WidgetImpl`:

```
template<typename T>
class WidgetImpl { ... };

template<typename T>
class Widget { ... };
```

Putting a `swap` member function in `Widget` (and, if we need to, in `WidgetImpl`) is as easy as before, but we run into trouble with the specialization for `std::swap`. This is what we want to write:

[Click here to view code image]

```
namespace std {
  template<typename T>
  void swap<Widget<T> >(Widget<T>& a,      // error! illegal
code!
                        Widget<T>& b)
  { a.swap(b); }
}
```

This looks perfectly reasonable, but it's not legal. We're trying to partially specialize a function template (`std::swap`), but though C++ allows partial specialization of class templates, it doesn't allow it for function templates. This code should not compile (though some compilers erroneously accept it).

When you want to “partially specialize” a function template, the usual approach is to simply add an overload. That would look like this:

[Click here to view code image]

```
namespace std {

  template<typename T>                // an overloading of
std::swap
  void swap(Widget<T>& a,             // (note the lack of "
<...>" after
            Widget<T>& b)             // "swap"), but see below
for
  { a.swap(b); }                      // why this isn't valid
code
}
```

In general, overloading function templates is fine, but `std` is a special namespace, and the rules governing it are special, too. It's okay to totally specialize templates in `std`, but it's not okay to add *new* templates (or classes or functions or anything else) to `std`. The contents of `std` are determined solely by the C++ standardization committee, and we're prohibited from augmenting what they've decided should go there. Alas, the form of the prohibition may dismay you. Programs that cross this line will almost certainly compile and run, but their behavior is undefined. If you want your software to have predictable behavior, you'll not add new things to `std`.

So what to do? We still need a way to let other people call `swap` and get our more efficient template-specific version. The answer is simple. We still declare a non-member `swap` that calls the member `swap`, we just don't declare the non-member to be a specialization or overloading of `std::swap`. For example, if all our `Widget`-related functionality is in the namespace `WidgetStuff`, it would look like this:

Click here to view code image

```
namespace WidgetStuff {
  ...                                     // templatized
WidgetImpl, etc.

  template<typename T>                    // as before,
including the swap
  class Widget { ... };                   // member function

  ...

  template<typename T>                    // non-member swap
function;
  void swap(Widget<T>& a,                 // not part of the
std namespace
            Widget<T>&
b)
  {
    a.swap(b);
  }
}
```

Now, if any code anywhere calls `swap` on two `Widget` objects, the name lookup rules in C++ (specifically the rules known as *argument-dependent lookup* or *Koenig lookup*) will find the `Widget`-specific version in `WidgetStuff`. Which is exactly what we want.

This approach works as well for classes as for class templates, so it seems like we should use it all the time. Unfortunately, there is a reason for specializing `std::swap` for classes (I'll describe it shortly), so if you want to have your class-specific version of `swap` called in as many contexts as possible (and you do), you need to write both a non-member version in the same namespace as your class and a specialization of `std::swap`.

By the way, if you're not using namespaces, everything above continues to apply (i.e., you still need a non-member `swap` that calls the member `swap`), but why are you clogging the global namespace with all your class, template, function, enum, enumerant, and typedef names? Have you no sense of propriety?

Everything I've written so far pertains to authors of `swap`, but it's worth looking at one situation from a client's point of view. Suppose you're writing a function template where you need to swap the values of two objects:

```
template<typename T>
void doSomething(T& obj1, T& obj2)
{
  ...
  swap(obj1, obj2);
  ...
}
```

Which `swap` should this call? The general one in `std`, which you know exists; a specialization of the general one in `std`, which may or may not exist; or a `T`-specific one, which may or may not exist and which may or may not be in a namespace (but should certainly not be in `std`)? What you desire is to call a `T`-specific version if there is one, but to fall back on the general version in `std` if there's not. Here's how you fulfill your desire:

Click here to view code image

```
template<typename T>
void doSomething(T& obj1, T& obj2)
{
  using std::swap;           // make std::swap available in
this function
  ...
  swap(obj1, obj2);          // call the best swap for
objects of type T
  ...
}
```

When compilers see the call to `swap`, they search for the right `swap` to invoke. C++'s name lookup rules ensure that this will find any `T`-specific `swap` at global scope or in the same namespace as the type `T`. (For example, if `T` is

`Widget` in the namespace `WidgetStuff`, compilers will use argument-dependent lookup to find `swap` in `WidgetStuff`.) If no `T`-specific `swap` exists, compilers will use `swap` in `std`, thanks to the `using` declaration that makes `std::swap` visible in this function. Even then, however, compilers will prefer a `T`-specific specialization of `std::swap` over the general template, so if `std::swap` has been specialized for `T`, the specialized version will be used.

Getting the right `swap` called is therefore easy. The one thing you want to be careful of is to not qualify the call, because that will affect how C++ determines the function to invoke. For example, if you were to write the call to `swap` this way,

```
std::swap(obj1, obj2);         // the wrong way to call swap
```

you'd force compilers to consider only the `swap` in `std` (including any template specializations), thus eliminating the possibility of getting a more appropriate `T`-specific version defined elsewhere. Alas, some misguided programmers *do* qualify calls to `swap` in this way, and that's why it's important to totally specialize `std::swap` for your classes: it makes type-specific `swap` implementations available to code written in this misguided fashion. (Such code is present in some standard library implementations, so it's in your interest to help such code work as efficiently as possible.)

At this point, we've discussed the default `swap`, member `swap`s, non-member `swap`s, specializations of `std::swap`, and calls to `swap`, so let's summarize the situation.

First, if the default implementation of `swap` offers acceptable efficiency for your class or class template, you don't need to do anything. Anybody trying to `swap` objects of your type will get the default version, and that will work fine.

Second, if the default implementation of `swap` isn't efficient enough (which almost always means that your class or template is using some variation of the pimpl idiom), do the following:

1. Offer a public `swap` member function that efficiently swaps the value of two objects of your type. For reasons I'll explain in a moment, this function should never throw an exception.
2. Offer a non-member `swap` in the same namespace as your class or

template. Have it call your `swap` member function.

3. If you're writing a class (not a class template), specialize `std::swap` for your class. Have it also call your `swap` member function.

Finally, if you're calling `swap`, be sure to include a `using` declaration to make `std::swap` visible in your function, then call `swap` without any namespace qualification.

The only loose end is my admonition to have the member version of `swap` never throw exceptions. That's because one of the most useful applications of `swap` is to help classes (and class templates) offer the strong exception-safety guarantee. Item 29 provides all the details, but the technique is predicated on the assumption that the member version of `swap` never throws. This constraint applies only to the member version! It can't apply to the non-member version, because the default version of `swap` is based on copy construction and copy assignment, and, in general, both of those functions are allowed to throw exceptions. When you write a custom version of `swap`, then, you are typically offering more than just an efficient way to swap values; you're also offering one that doesn't throw exceptions. As a general rule, these two `swap` characteristics go hand in hand, because highly efficient `swap`s are almost always based on operations on built-in types (such as the pointers underlying the pimpl idiom), and operations on built-in types never throw exceptions.

Things to Remember

- Provide a `swap` member function when `std::swap` would be inefficient for your type. Make sure your `swap` doesn't throw exceptions.
- If you offer a member `swap`, also offer a non-member `swap` that calls the member. For classes (not templates), specialize `std::swap`, too.
- When calling `swap`, employ a `using` declaration for `std::swap`, then call `swap` without namespace qualification.
- It's fine to totally specialize `std` templates for user-defined types, but never try to add something completely new to `std`.

5. Implementations

For the most part, coming up with appropriate definitions for your classes (and class templates) and appropriate declarations for your functions (and function templates) is the lion's share of the battle. Once you've got those right, the corresponding implementations are largely straightforward. Still, there are things to watch out for. Defining variables too soon can cause a drag on performance. Overuse of casts can lead to code that's slow, hard to maintain, and infected with subtle bugs. Returning handles to an object's internals can defeat encapsulation and leave clients with dangling handles. Failure to consider the impact of exceptions can lead to leaked resources and corrupted data structures. Overzealous inlining can cause code bloat. Excessive coupling can result in unacceptably long build times.

All of these problems can be avoided. This chapter explains how.

Item 26: Postpone variable definitions as long as possible.

Whenever you define a variable of a type with a constructor or destructor, you incur the cost of construction when control reaches the variable's definition, and you incur the cost of destruction when the variable goes out of scope. There's a cost associated with unused variables, so you want to avoid them whenever you can.

You're probably thinking that you never define unused variables, but you may need to think again. Consider the following function, which returns an encrypted version of a password, provided the password is long enough. If the password is too short, the function throws an exception of type `logic_error`, which is defined in the standard C++ library (see Item 54):

Click here to view code image

```
// this function defines the variable "encrypted" too soon
std::string encryptPassword(const std::string& password)
{
  using namespace std;
  string encrypted;

  if (password.length() < MinimumPasswordLength) {
      throw logic_error("Password is too short");
  }
  ...                             // do whatever is necessary to
place an
                                  // encrypted version of password
in encrypted
  return encrypted;
}
```

The object `encrypted` isn't *completely* unused in this function, but it's unused if an exception is thrown. That is, you'll pay for the construction and destruction of `encrypted` even if `encryptPassword` throws an exception. As a result, you're better off postponing `encrypted`'s definition until you *know* you'll need it:

Click here to view code image

```
// this function postpones encrypted's definition until it's
truly necessary
std::string encryptPassword(const std::string& password)
{
  using namespace std;

  if (password.length() < MinimumPasswordLength) {
     throw logic_error("Password is too short");
  }

  string encrypted;

  ...                           // do whatever is necessary to
place an
                                // encrypted version of password
in encrypted
  return encrypted;
}
```

This code still isn't as tight as it might be, because `encrypted` is defined without any initialization arguments. That means its default constructor will be used. In many cases, the first thing you'll do to an object is give it some value, often via an assignment. Item 4 explains why default-constructing an object and then assigning to it is less efficient than initializing it with the value you really want it to have. That analysis applies here, too. For example, suppose the hard part of `encryptPassword` is performed in this function:

```
void encrypt(std::string& s);             // encrypts s in
place
```

Then `encryptPassword` could be implemented like this, though it wouldn't be the best way to do it:

Click here to view code image

```
// this function postpones encrypted's definition until
// it's necessary, but it's still needlessly inefficient
std::string encryptPassword(const std::string& password)
{
  ...                                   // import std and
check length as above

  string encrypted;                     // default-construct
encrypted
  encrypted = password;                 // assign to
encrypted

  encrypt(encrypted);
  return encrypted;
}
```

A preferable approach is to initialize `encrypted` with `password`, thus skipping the pointless and potentially expensive default construction:

Click here to view code image

```
// finally, the best way to define and initialize encrypted
std::string encryptPassword(const std::string& password)
{
  ...                                     // import std and
check length

  string encrypted(password);             // define and
initialize via copy
                                          // constructor

  encrypt(encrypted);
  return encrypted;
}
```

This suggests the real meaning of “as long as possible” in this Item's title. Not only should you postpone a variable's definition until right before you have to use the variable, you should also try to postpone the definition until you have initialization arguments for it. By doing so, you avoid constructing and destructing unneeded objects, and you avoid unnecessary default constructions. Further, you help document the purpose of variables by initializing them in contexts in which their meaning is clear.

“But what about loops?” you may wonder. If a variable is used only inside a loop, is it better to define it outside the loop and make an assignment to it on each loop iteration, or is it be better to define the variable inside the loop? That is, which of these general structures is better?

Click here to view code image

```
// Approach A: define outside loop   // Approach B: define
inside loop

Widget w;
for (int i = 0; i < n; ++i) {         for (int i = 0; i < n;
++i) {
  w = some value dependent on i;       Widget w(some value
dependent on i);
  ...                                   ...
}                                    }
```

Here I've switched from an object of type `string` to an object of type `Widget` to avoid any preconceptions about the cost of performing a construction, destruction, or assignment for the object.

In terms of `Widget` operations, the costs of these two approaches are as follows:

- Approach A: 1 constructor + 1 destructor + `n` assignments.
- Approach B: `n` constructors + `n` destructors.

For classes where an assignment costs less than a constructor-destructor pair, Approach A is generally more efficient. This is especially the case as `n` gets large. Otherwise, Approach B is probably better. Furthermore, Approach A makes the name `w` visible in a larger scope (the one containing the loop) than

Approach B, something that's contrary to program comprehensibility and maintainability. As a result, unless you know that (1) assignment is less expensive than a constructor-destructor pair and (2) you're dealing with a performance-sensitive part of your code, you should default to using Approach B.

Things to Remember

- Postpone variable definitions as long as possible. It increases program clarity and improves program efficiency.

Item 27: Minimize casting.

The rules of C++ are designed to guarantee that type errors are impossible. In theory, if your program compiles cleanly, it's not trying to perform any unsafe or nonsensical operations on any objects. This is a valuable guarantee. You don't want to forgo it lightly.

Unfortunately, casts subvert the type system. That can lead to all kinds of trouble, some easy to recognize, some extraordinarily subtle. If you're coming to C++ from C, Java, or C#, take note, because casting in those languages is more necessary and less dangerous than in C++. But C++ is not C. It's not Java. It's not C#. In this language, casting is a feature you want to approach with great respect.

Let's begin with a review of casting syntax, because there are usually three different ways to write the same cast. C-style casts look like this:

```
(T) expression                      // cast expression to be
of type T
```

Function-style casts use this syntax:

```
T(expression)                       // cast expression to be
of type T
```

There is no difference in meaning between these forms; it's purely a matter of where you put the parentheses. I call these two forms *old-style casts*.

C++ also offers four new cast forms (often called *new-style* or *C++-style casts*):

```
const_cast<T>(expression)
dynamic_cast<T>(expression)
reinterpret_cast<T>(expression)
static_cast<T>(expression)
```

Each serves a distinct purpose:

- `const_cast` is typically used to cast away the constness of objects. It is the only C++-style cast that can do this.
- `dynamic_cast` is primarily used to perform “safe downcasting,” i.e., to determine whether an object is of a particular type in an inheritance hierarchy. It is the only cast that cannot be performed using the old-style syntax. It is also the only cast that may have a significant runtime cost. (I'll provide details on this a bit later.)
- `reinterpret_cast` is intended for low-level casts that yield implementation-dependent (i.e., unportable) results, e.g., casting a pointer to an `int`. Such casts should be rare outside low-level code. I use it only once in this book, and that's only when discussing how you might write a debugging allocator for raw memory (see Item 50).
- `static_cast` can be used to force implicit conversions (e.g., non-`const` object to `const` object (as in Item 3), `int` to `double`, etc.). It can also be used to perform the reverse of many such conversions (e.g., `void*` pointers to typed pointers, pointer-to-base to pointer-to-derived), though it cannot cast from `const` to non-`const` objects. (Only `const_cast` can do that.)

The old-style casts continue to be legal, but the new forms are preferable. First, they're much easier to identify in code (both for humans and for tools like `grep`), thus simplifying the process of finding places in the code where the type system is being subverted. Second, the more narrowly specified purpose of each cast makes it possible for compilers to diagnose usage errors. For example, if you try to cast away constness using a new-style cast other than `const_cast`, your code won't compile.

About the only time I use an old-style cast is when I want to call an `explicit` constructor to pass an object to a function. For example:

Click here to view code image

```
class Widget {
```

```
public:
  explicit Widget(int size);
  ...
};

void doSomeWork(const Widget& w);

doSomeWork(Widget(15));                       // create Widget
from int
                                              // with function-
style cast

doSomeWork(static_cast<Widget>(15));          // create Widget
from int
                                              // with C++-style
cast
```

Somehow, deliberate object creation doesn't “feel” like a cast, so I'd probably use the function-style cast instead of the `static_cast` in this case. (They do exactly the same thing here: create a temporary Widget object to pass to doSomeWork.) Then again, code that leads to a core dump usually feels pretty reasonable when you write it, so perhaps you'd best ignore feelings and use new-style casts all the time.

Many programmers believe that casts do nothing but tell compilers to treat one type as another, but this is mistaken. Type conversions of any kind (either explicit via casts or implicit by compilers) often lead to code that is executed at runtime. For example, in this code fragment,

Click here to view code image

```
int x, y;
...
double d = static_cast<double>(x)/y;           // divide x by
y, but use
                                               // floating
point division
```

the cast of the `int x` to a `double` almost certainly generates code, because on most architectures, the underlying representation for an `int` is different from that for a `double`. That's perhaps not so surprising, but this example may widen your eyes a bit:

Click here to view code image

```
class Base { ... };

class Derived: public Base { ... };

Derived d;

Base *pb = &d;                         // implicitly convert
Derived* ⇒ Base*
```

Here we're just creating a base class pointer to a derived class object, but sometimes, the two pointer values will not be the same. When that's the case, an offset is applied *at runtime* to the `Derived*` pointer to get the correct `Base*` pointer value.

This last example demonstrates that a single object (e.g., an object of type `Derived`) might have more than one address (e.g., its address when pointed to by a `Base*` pointer and its address when pointed to by a `Derived*` pointer). That can't happen in C. It can't happen in Java. It can't happen in C#. It *does* happen in C++. In fact, when multiple inheritance is in use, it happens virtually all the time, but it can happen under single inheritance, too. Among other things, that means you should generally avoid making assumptions about how things are laid out in C++, and you should certainly not perform casts based on such assumptions. For example, casting object addresses to `char*` pointers and then using pointer arithmetic on them almost always yields undefined behavior.

But note that I said that an offset is “sometimes” required. The way objects are laid out and the way their addresses are calculated varies from compiler to compiler. That means that just because your “I know how things are laid out” casts work on one platform doesn't mean they'll work on others. The world is filled with woeful programmers who've learned this lesson the hard way.

An interesting thing about casts is that it's easy to write something that looks right (and might be right in other languages) but is wrong. Many application frameworks, for example, require that virtual member function implementations in derived classes call their base class counterparts first. Suppose we have a `Window` base class and a `SpecialWindow` derived class, both of which define the virtual function `onResize`. Further suppose that `SpecialWindow`'s `onResize` is expected to invoke `Window`'s `onResize` first. Here's a way to implement this that looks like it does the right thing, but doesn't:

Click here to view code image

```
class Window {                              // base class
public:
  virtual void onResize() { ... }           // base
onResize impl
  ...
};

class SpecialWindow: public Window {        // derived
class
public:
  virtual void onResize() {                 // derived
onResize impl;
    static_cast<Window>(*this).onResize();  // cast *this
to Window,
                                            // then call
its onResize;
                                            // this doesn't
work!

    ...                                     // do
SpecialWindow-
  }                                         // specific
stuff

  ...

};
```

I've highlighted the cast in the code. (It's a new-style cast, but using an old-style cast wouldn't change anything.) As you would expect, the code casts `*this` to a `Window`. The resulting call to `onResize` therefore invokes `Window::onResize`. What you might not expect is that it does not invoke that function on the current object! Instead, the cast creates a new, temporary *copy* of the base class part of `*this`, then invokes `onResize` on the copy! The above code doesn't call `Window::onResize` on the current object and then perform the `SpecialWindow`-specific actions on that object — it calls `Window::onResize` on a *copy of the base class part* of the current object before performing `SpecialWindow`-specific actions on the current object. If `Window::onResize` modifies the current object (hardly a remote possibility, since `onResize` is a non-`const` member function), the current object won't be modified. Instead, a *copy* of that object will be modified. If `SpecialWindow::onResize` modifies the current object, however, the current object *will* be modified, leading to the prospect that the code will leave the current object in an invalid state, one where base class modifications have not been made, but derived class ones have been.

The solution is to eliminate the cast, replacing it with what you really want to say. You don't want to trick compilers into treating `*this` as a base class object; you want to call the base class version of `onResize` on the current object. So say that:

Click here to view code image

```
class SpecialWindow: public Window {
public:
  virtual void onResize() {
    Window::onResize();                          // call
Window::onResize
    ...                                          // on *this
  }
  ...

};
```

This example also demonstrates that if you find yourself wanting to cast, it's a sign that you could be approaching things the wrong way. This is especially the case if your want is for `dynamic_cast`.

Before delving into the design implications of `dynamic_cast`, it's worth observing that many implementations of `dynamic_cast` can be quite slow. For example, at least one common implementation is based in part on string comparisons of class names. If you're performing a `dynamic_cast` on an object in a single-inheritance hierarchy four levels deep, each `dynamic_cast` under such an implementation could cost you up to four calls to `strcmp` to compare class names. A deeper hierarchy or one using multiple inheritance would be more expensive. There are reasons that some implementations work this way (they have to do with support for dynamic linking). Nonetheless, in addition to being leery of casts in general, you should be especially leery of `dynamic_cast`s in performance-sensitive code.

The need for `dynamic_cast` generally arises because you want to perform derived class operations on what you believe to be a derived class object, but you have only a pointer- or reference-to-base through which to manipulate the object. There are two general ways to avoid this problem.

First, use containers that store pointers (often smart pointers — see Item 13) to derived class objects directly, thus eliminating the need to manipulate such objects through base class interfaces. For example, if, in our `Window`/`SpecialWindow` hierarchy, only `SpecialWindow`s support blinking, instead of doing this:

Click here to view code image

```
class Window { ... };

class SpecialWindow: public Window {
public:
  void blink();
  ...
};
typedef                                           // see
Item 13 for info
  std::vector<std::tr1::shared_ptr<Window> > VPW;  // on
tr1::shared_ptr

VPW winPtrs;

...

for (VPW::iterator iter = winPtrs.begin();         //
undesirable code:
     iter != winPtrs.end();                       // uses
dynamic_cast
     ++iter) {
  if (SpecialWindow *psw = dynamic_cast<SpecialWindow*>(iter-
>get()))
     psw->blink();
}
```

try to do this instead:

Click here to view code image

```
typedef std::vector<std::tr1::shared_ptr<SpecialWindow> >
VPSW;
```

```
VPSW winPtrs;

...

for (VPSW::iterator iter = winPtrs.begin();        // better
code: uses
     iter != winPtrs.end();                        // no
dynamic_cast
     ++iter)
  (*iter)->blink();
```

Of course, this approach won't allow you to store pointers to all possible `Window` derivatives in the same container. To work with different window types, you might need multiple type-safe containers.

An alternative that will let you manipulate all possible `Window` derivatives through a base class interface is to provide virtual functions in the base class that let you do what you need. For example, though only `SpecialWindow`s can blink, maybe it makes sense to declare the function in the base class, offering a default implementation that does nothing:

Click here to view code image

```
class Window {
public:
  virtual void blink() {}                       // default
impl is no-op;
  ...                                           // see Item
34 for why
};                                              // a default
impl may be
                                                // a bad idea

class SpecialWindow: public Window {
public:
  virtual void blink() { ... };                 // in this
class, blink
  ...                                           // does
something
};

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;

VPW winPtrs;                                    // container
```

```
holds
                                                // (ptrs to)
all possible
...                                             // Window
types

for (VPW::iterator iter = winPtrs.begin();
     iter != winPtrs.end();
     ++iter)                                    // note lack
of
  (*iter)->blink();                             //
dynamic_cast
```

Neither of these approaches — using type-safe containers or moving virtual functions up the hierarchy — is universally applicable, but in many cases, they provide a viable alternative to `dynamic_cast`ing. When they do, you should embrace them.

One thing you definitely want to avoid is designs that involve cascading `dynamic_cast`s, i.e., anything that looks like this:

Click here to view code image

```
class Window { ... };

...                                     // derived classes
are defined here

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;

VPW winPtrs;

...

for (VPW::iterator iter = winPtrs.begin(); iter !=
winPtrs.end(); ++iter)
{
  if (SpecialWindow1 *psw1 =
       dynamic_cast<SpecialWindow1*>(iter->get())) { ... }

  else if (SpecialWindow2 *psw2 =
            dynamic_cast<SpecialWindow2*>(iter->get())) { ...
}

  else if (SpecialWindow3 *psw3 =
            dynamic_cast<SpecialWindow3*>(iter->get())) { ...
}

  ...
}
```

Such C++ generates code that's big and slow, plus it's brittle, because every time the `Window` class hierarchy changes, all such code has to be examined to see if it needs to be updated. (For example, if a new derived class gets added, a new conditional branch probably needs to be added to the above cascade.) Code that looks like this should almost always be replaced with something based on virtual function calls.

Good C++ uses very few casts, but it's generally not practical to get rid of all of them. The cast from `int` to `double` on page 118, for example, is a reasonable use of a cast, though it's not strictly necessary. (The code could be rewritten to declare a new variable of type `double` that's initialized with `x`'s value.) Like most suspicious constructs, casts should be isolated as much as possible, typically hidden inside functions whose interfaces shield callers from the grubby work being done inside.

Things to Remember

- Avoid casts whenever practical, especially `dynamic_cast`s in performance-sensitive code. If a design requires casting, try to develop a cast-free alternative.
- When casting is necessary, try to hide it inside a function. Clients can then call the function instead of putting casts in their own code.
- Prefer C++-style casts to old-style casts. They are easier to see, and they are more specific about what they do.

Item 28: Avoid returning “handles” to object internals.

Suppose you're working on an application involving rectangles. Each rectangle can be represented by its upper left corner and its lower right corner. To keep a `Rectangle` object small, you might decide that the points defining its extent shouldn't be stored in the `Rectangle` itself, but rather in an auxiliary struct that the `Rectangle` points to:

Click here to view code image

```
class Point {                      // class for representing
points
public:
  Point(int x, int y);
  ...

  void setX(int newVal);
  void setY(int newVal);
  ...
};

struct RectData {                    // Point data for a
Rectangle
  Point ulhc;                        // ulhc = " upper left-
hand corner"
  Point lrhc;                        // lrhc = " lower right-
hand corner"
};

class Rectangle {
  ...

private:
  std::tr1::shared_ptr<RectData> pData;          // see Item
13 for info on
};                                               //
tr1::shared_ptr
```

Because `Rectangle` clients will need to be able to determine the extent of a `Rectangle`, the class provides the `upperLeft` and `lowerRight` functions. However, `Point` is a user-defined type, so, mindful of Item 20's observation that passing user-defined types by reference is typically more efficient than passing them by value, these functions return references to the underlying `Point` objects:

Click here to view code image

```
class Rectangle {
public:
  ...
  Point& upperLeft() const { return pData->ulhc; }
  Point& lowerRight() const { return pData->lrhc; }
  ...
};
```

This design will compile, but it's wrong. In fact, it's self-contradictory. On the one hand, `upperLeft` and `lowerRight` are declared to be `const` member functions, because they are designed only to offer clients a way to learn what the `Rectangle`'s points are, not to let clients modify the `Rectangle` (see Item 3). On the other hand, both functions return references to private internal data — references that callers can use to modify that internal data! For example:

Click here to view code image

```
Point coord1(0, 0);
Point coord2(100, 100);

const Rectangle rec(coord1, coord2);     // rec is a const
rectangle from
                                         // (0, 0) to (100,
100)

rec.upperLeft().setX(50);                // now rec goes from
                                         // (50, 0) to (100,
100)!
```

Here, notice how the caller of `upperLeft` is able to use the returned reference to one of `rec`'s internal `Point` data members to modify that member. But `rec` is supposed to be `const`!

This immediately leads to two lessons. First, a data member is only as encapsulated as the most accessible function returning a reference to it. In this case, though `ulhc` and `lrhc` are supposed to be private to their `Rectangle`, they're effectively public, because the public functions `upperLeft` and `lowerRight` return references to them. Second, if a `const` member function returns a reference to data associated with an object that is stored outside the object itself, the caller of the function can modify that data. (This is just a fallout of the limitations of bitwise constness — see Item 3.)

Everything we've done has involved member functions returning references, but if they returned pointers or iterators, the same problems would exist for the same reasons. References, pointers, and iterators are all *handles* (ways to get at other objects), and returning a handle to an object's internals always runs the risk of compromising an object's encapsulation. As we've seen, it can also lead to `const` member functions that allow an object's state to be modified.

We generally think of an object's “internals” as its data members, but member functions not accessible to the general public (i.e., that are protected or private) are part of an object's internals, too. As such, it's important not to return handles to them. This means you should never have a member function return a pointer to a less accessible member function. If you do, the effective access level will be that of the more accessible function, because clients will be able to get a pointer to the less accessible function, then call that function through the pointer.

Functions that return pointers to member functions are uncommon, however, so let's turn our attention back to the `Rectangle` class and its `upperLeft` and `lowerRight` member functions. Both of the problems we've identified for those functions can be eliminated by simply applying `const` to their return types:

Click here to view code image

```
class Rectangle {
public:
  ...
  const Point& upperLeft() const { return pData->ulhc; }
  const Point& lowerRight() const { return pData->lrhc; }
  ...
};
```

With this altered design, clients can read the `Point`s defining a rectangle, but they can't write them. This means that declaring `upperLeft` and `lowerRight` as `const` is no longer a lie, because they no longer allow callers to modify the state of the object. As for the encapsulation problem, we always intended to let clients see the `Point`s making up a `Rectangle`, so this is a deliberate relaxation of encapsulation. More importantly, it's a *limited* relaxation: only read access is being granted by these functions. Write access is still prohibited.

Even so, `upperLeft` and `lowerRight` are still returning handles to an object's internals, and that can be problematic in other ways. In particular, it can lead to *dangling handles*: handles that refer to parts of objects that don't exist any longer. The most common source of such disappearing objects are function return values. For example, consider a function that returns the bounding box for a GUI object in the form of a rectangle:

Click here to view code image

```
class GUIObject { ... };

const Rectangle                             // returns a
rectangle by
  boundingBox(const GUIObject& obj);        // value; see
Item 3 for why
                                            // return type is
const
```

Now consider how a client might use this function:

Click here to view code image

```
GUIObject *pgo;                             // make pgo point
to
...                                         // some GUIObject

const Point *pUpperLeft =                   // get a ptr to
the upper
  &(boundingBox(*pgo).upperLeft());         // left point of
its
                                            // bounding box
```

The call to `boundingBox` will return a new, temporary `Rectangle` object. That object doesn't have a name, so let's call it *temp*. `upperLeft` will then be called on *temp*, and that call will return a reference to an internal part of *temp*, in particular, to one of the `Point`s making it up. `pUpperLeft` will then point to that `Point` object. So far, so good, but we're not done yet, because at the end of the statement, `boundingBox`'s return value — *temp* — will be destroyed, and that will indirectly lead to the destruction of *temp*'s `Point`s. That, in turn, will leave `pUpperLeft` pointing to an object that no longer exists; `pUpperLeft` will dangle by the end of the statement that created it!

This is why any function that returns a handle to an internal part of the object is dangerous. It doesn't matter whether the handle is a pointer, a reference, or an iterator. It doesn't matter whether it's qualified with `const`. It doesn't matter whether the member function returning the handle is itself `const`. All that matters is that a handle is being returned, because once that's being done, you run the risk that the handle will outlive the object it refers to.

This doesn't mean that you should *never* have a member function that returns a handle. Sometimes you have to. For example, `operator[]` allows you to pluck individual elements out of `string`s and `vector`s, and these `operator[]`s work by returning references to the data in the containers (see Item 3) — data that is destroyed when the containers themselves are. Still, such functions are the exception, not the rule.

Things to Remember

- Avoid returning handles (references, pointers, or iterators) to object internals. Not returning handles increases encapsulation, helps `const` member functions act `const`, and minimizes the creation of dangling handles.

Item 29: Strive for exception-safe code.

Exception safety is sort of like pregnancy...but hold that thought for a moment. We can't really talk reproduction until we've worked our way through courtship.

Suppose we have a class for representing GUI menus with background images. The class is designed to be used in a threaded environment, so it has a mutex for concurrency control:

Click here to view code image

```
class PrettyMenu {
public:
  ...
  void changeBackground(std::istream& imgSrc);           //
change background
  ...                                                    //
image

private:

  Mutex mutex;                         // mutex for this object

  Image *bgImage;                      // current background image
  int imageChanges;                    // # of times image has
been changed
};
```

Consider this possible implementation of `PrettyMenu`'s `changeBackground` function:

Click here to view code image

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  lock(&mutex);                      // acquire mutex (as in
Item 14)

  delete bgImage;                    // get rid of old
background
  ++imageChanges;                    // update image change
count
  bgImage = new Image(imgSrc);       // install new
background

  unlock(&mutex);                    // release mutex
}
```

From the perspective of exception safety, this function is about as bad as it gets. There are two requirements for exception safety, and this satisfies neither.

When an exception is thrown, exception-safe functions:

- **Leak no resources.** The code above fails this test, because if the "`new Image(imgSrc)`" expression yields an exception, the call to `unlock` never gets executed, and the mutex is held forever.
- **Don't allow data structures to become corrupted.** If "`new Image(imgSrc)`" throws, `bgImage` is left pointing to a deleted object. In addition, `imageChanges` has been incremented, even though it's not true that a new image has been installed. (On the other hand, the old image has definitely been eliminated, so I suppose you could argue that the image has been "changed.")

Addressing the resource leak issue is easy, because Item 13 explains how to use objects to manage resources, and Item 14 introduces the `Lock` class as a way to ensure that mutexes are released in a timely fashion:

Click here to view code image

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  Lock ml(&mutex);                // from Item 14: acquire
mutex and
```

```
                                    // ensure its later
release
  delete bgImage;
  ++imageChanges;
  bgImage = new Image(imgSrc);
}
```

One of the best things about resource management classes like `Lock` is that they usually make functions shorter. See how the call to `unlock` is no longer needed? As a general rule, less code is better code, because there's less to go wrong and less to misunderstand when making changes.

With the resource leak behind us, we can turn our attention to the issue of data structure corruption. Here we have a choice, but before we can choose, we have to confront the terminology that defines our choices.

Exception-safe functions offer one of three guarantees:

- Functions offering **the basic guarantee** promise that if an exception is thrown, everything in the program remains in a valid state. No objects or data structures become corrupted, and all objects are in an internally consistent state (e.g., all class invariants are satisfied). However, the exact state of the program may not be predictable. For example, we could write `changeBackground` so that if an exception were thrown, the `PrettyMenu` object might continue to have the old background image, or it might have some default background image, but clients wouldn't be able to predict which. (To find out, they'd presumably have to call some member function that would tell them what the current background image was.)
- Functions offering **the strong guarantee** promise that if an exception is thrown, the state of the program is unchanged. Calls to such functions are *atomic* in the sense that if they succeed, they succeed completely, and if they fail, the program state is as if they'd never been called.

  Working with functions offering the strong guarantee is easier than working with functions offering only the basic guarantee, because after calling a function offering the strong guarantee, there are only two possible program states: as expected following successful execution of the function, or the state that existed at the time the function was called. In contrast, if a call to a function offering only the basic guarantee yields an exception, the program could be in *any* valid state.
- Functions offering **the nothrow guarantee** promise never to throw

exceptions, because they always do what they promise to do. All operations on built-in types (e.g., `int`s, pointers, etc.) are nothrow (i.e., offer the nothrow guarantee). This is a critical building block of exception-safe code.

It might seem reasonable to assume that functions with an empty exception specification are nothrow, but this isn't necessarily true. For example, consider this function:

```
int doSomething() throw();          // note empty
exception spec.
```

This doesn't say that `doSomething` will never throw an exception; it says that *if* `doSomething` throws an exception, it's a serious error, and the `unexpected` function should be called.[†] In fact, `doSomething` may not offer any exception guarantee at all. The declaration of a function (including its exception specification, if it has one) doesn't tell you whether a function is correct or portable or efficient, and it doesn't tell you which, if any, exception safety guarantee it offers, either. All those characteristics are determined by the function's implementation, not its declaration.

Exception-safe code must offer one of the three guarantees above. If it doesn't, it's not exception-safe. The choice, then, is to determine which guarantee to offer for each of the functions you write. Other than when dealing with exception-unsafe legacy code (which we'll discuss later in this Item), offering no exception safety guarantee should be an option only if your crack team of requirements analysts has identified a need for your application to leak resources and run with corrupt data structures.

As a general rule, you want to offer the strongest guarantee that's practical. From an exception safety point of view, nothrow functions are wonderful, but it's hard to climb out of the C part of C++ without calling functions that might throw. Anything using dynamically allocated memory (e.g., all STL containers) typically throws a `bad_alloc` exception if it can't find enough memory to satisfy a request (see Item 49). Offer the nothrow guarantee when you can, but for most functions, the choice is between the basic and strong guarantees.

In the case of `changeBackground`, *almost* offering the strong guarantee is not difficult. First, we change the type of `PrettyMenu`'s `bgImage` data member from a built-in `Image*` pointer to one of the smart resource-managing pointers described in Item 13. Frankly, this is a good idea purely on the basis of preventing resource leaks. The fact that it helps us offer the strong exception safety guarantee simply reinforces Item 13's argument that using objects (such as smart pointers) to manage resources is fundamental to good design. In the code below, I show use of `tr1::shared_ptr`, because its more intuitive behavior when copied generally makes it preferable to `auto_ptr`.

Second, we reorder the statements in `changeBackground` so that we don't increment `imageChanges` until the image has been changed. As a general rule, it's a good policy not to change the status of an object to indicate that something has happened until something actually has.

Here's the resulting code:

Click here to view code image

```
class PrettyMenu {
  ...
  std::tr1::shared_ptr<Image> bgImage;
  ...
};

void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  Lock ml(&mutex);

  bgImage.reset(new Image(imgSrc));  // replace bgImage's
internal
                                     // pointer with the
result of the
                                     // "new Image"
expression
  ++imageChanges;
}
```

Note that there's no longer a need to manually delete the old image, because that's handled internally by the smart pointer. Furthermore, the deletion takes place only if the new image is successfully created. More precisely, the `tr1::shared_ptr::reset` function will be called only if its parameter (the result of “`new Image(imgSrc)`”) is successfully created. `delete` is used only inside the call to `reset`, so if the function is never entered, `delete` is never used. Note also that the use of an object (the `tr1::shared_ptr`) to manage a resource (the dynamically allocated `Image`) has again pared the length of `changeBackground`.

As I said, those two changes *almost* suffice to allow `changeBackground` to offer the strong exception safety guarantee. What's the fly in the ointment? The parameter `imgSrc`. If the `Image` constructor throws an exception, it's possible that the read marker for the input stream has been moved, and such movement would be a change in state visible to the rest of the program. Until `changeBackground` addresses that issue, it offers only the basic exception safety guarantee.

Let's set that aside, however, and pretend that `changeBackground` does offer the strong guarantee. (I'm confident you could come up with a way for it to do so, perhaps by changing its parameter type from an `istream` to the name of the file containing the image data.) There is a general design strategy that typically leads to the strong guarantee, and it's important to be familiar with it. The strategy is known as "copy and swap." In principle, it's very simple. Make a copy of the object you want to modify, then make all needed changes to the copy. If any of the modifying operations throws an exception, the original object remains unchanged. After all the changes have been successfully completed, swap the modified object with the original in a non-throwing operation.

This is usually implemented by putting all the per-object data from the "real" object into a separate implementation object, then giving the real object a pointer to its implementation object. This is often known as the "pimpl idiom," and Item 31 describes it in some detail. For `PrettyMenu`, it would typically look something like this:

Click here to view code image

```
struct PMImpl {                               // PMImpl =
"PrettyMenu
  std::tr1::shared_ptr<Image> bgImage;        // Impl."; see
below for
  int imageChanges;                           // why it's a
struct
};

class PrettyMenu {
  ...

private:
  Mutex mutex;
  std::tr1::shared_ptr<PMImpl> pImpl;
};

void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  using std::swap;                             // see Item 25

  Lock ml(&mutex);                             // acquire the
mutex

  std::tr1::shared_ptr<PMImpl>                 // copy obj.
data
    pNew(new PMImpl(*pImpl));

  pNew->bgImage.reset(new Image(imgSrc));      // modify the
copy
  ++pNew->imageChanges;

  swap(pImpl, pNew);                           // swap the new
                                               // data into
place

}                                              // release the
mutex
```

In this example, I've chosen to make `PMImpl` a struct instead of a class, because the encapsulation of `PrettyMenu` data is assured by `pImpl` being private. Making `PMImpl` a class would be at least as good, though somewhat less convenient. (It would also keep the object-oriented purists at bay.) If desired, `PMImpl` could be nested inside `PrettyMenu`, but packaging issues such as that are independent of writing exception-safe code, which is our concern here.

The copy-and-`swap` strategy is an excellent way to make all-or-nothing changes to an object's state, but, in general, it doesn't guarantee that the overall function is strongly exception-safe. To see why, consider an abstraction of `changeBackground`, `someFunc`, that uses copy-and-`swap`, but that includes calls to two other functions, `f1` and `f2`:

Click here to view code image

```
void someFunc()
{
  ...                                       // make copy of
local state
```

```
  f1();
  f2();
  ...                                     // swap modified
state into place
}
```

It should be clear that if `f1` or `f2` is less than strongly exception-safe, it will be hard for `someFunc` to be strongly exception-safe. For example, suppose that `f1` offers only the basic guarantee. For `someFunc` to offer the strong guarantee, it would have to write code to determine the state of the entire program prior to calling `f1`, catch all exceptions from `f1`, then restore the original state.

Things aren't really any better if both `f1` and `f2` *are* strongly exception safe. After all, if `f1` runs to completion, the state of the program may have changed in arbitrary ways, so if `f2` then throws an exception, the state of the program is not the same as it was when `someFunc` was called, even though `f2` didn't change anything.

The problem is side effects. As long as functions operate only on local state (e.g., `someFunc` affects only the state of the object on which it's invoked), it's relatively easy to offer the strong guarantee. When functions have side effects on non-local data, it's much harder. If a side effect of calling `f1`, for example, is that a database is modified, it will be hard to make `someFunc` strongly exception-safe. There is, in general, no way to undo a database modification that has already been committed; other database clients may have already seen the new state of the database.

Issues such as these can prevent you from offering the strong guarantee for a function, even though you'd like to. Another issue is efficiency. The crux of copy-and-`swap` is the idea of modifying a copy of an object's data, then swapping the modified data for the original in a non-throwing operation. This requires making a copy of each object to be modified, which takes time and space you may be unable or unwilling to make available. The strong guarantee is highly desirable, and you should offer it when it's practical, but it's not practical 100% of the time.

When it's not, you'll have to offer the basic guarantee. In practice, you'll probably find that you can offer the strong guarantee for some functions, but the cost in efficiency or complexity will make it untenable for many others. As long as you've made a reasonable effort to offer the strong guarantee whenever it's practical, no one should be in a position to criticize you when you offer only the basic guarantee. For many functions, the basic guarantee is a perfectly reasonable choice.

Things are different if you write a function offering no exception-safety guarantee at all, because in this respect it's reasonable to assume that you're guilty until proven innocent. You *should* be writing exception-safe code. But you may have a compelling defense. Consider again the implementation of `someFunc` that calls the functions `f1` and `f2`. Suppose `f2` offers no exception safety guarantee at all, not even the basic guarantee. That means that if `f2` emits an exception, the program may have leaked resources inside `f2`. It means that `f2` may have corrupted data structures, e.g., sorted arrays might not be sorted any longer, objects being transferred from one data structure to another might have been lost, etc. There's no way that `someFunc` can compensate for those problems. If the functions `someFunc` calls offer no exception-safety guarantees, `someFunc` itself can't offer any guarantees.

Which brings me back to pregnancy. A female is either pregnant or she's not. It's not possible to be partially pregnant. Similarly, a software system is either exception-safe or it's not. There's no such thing as a partially exception-safe system. If a system has even a single function that's not exception-safe, the system as a whole is not exception-safe, because calls to that one function could lead to leaked resources and corrupted data structures. Unfortunately, much C++ legacy code was written without exception safety in mind, so many systems today are not exception-safe. They incorporate code that was written in an exception-unsafe manner.

There's no reason to perpetuate this state of affairs. When writing new code or modifying existing code, think carefully about how to make it exception-safe. Begin by using objects to manage resources. (Again, see Item 13.) That will prevent resource leaks. Follow that by determining which of the three exception safety guarantees is the strongest you can practically offer for each function you write, settling for no guarantee only if calls to legacy code leave you no choice. Document your decisions, both for clients of your functions and for future maintainers. A function's exception-safety guarantee is a visible part of its interface, so you should choose it as deliberately as you choose all other aspects of a function's interface.

Forty years ago, `goto`-laden code was considered perfectly good practice. Now we strive to write structured control flows. Twenty years ago, globally accessible data was considered perfectly good practice. Now we strive to encapsulate data. Ten years ago, writing functions without thinking about the impact of exceptions was considered perfectly good practice. Now we strive to write exception-safe code.

Time goes on. We live. We learn.

Things to Remember

- Exception-safe functions leak no resources and allow no data structures to become corrupted, even when exceptions are thrown. Such functions offer the basic, strong, or nothrow guarantees.
- The strong guarantee can often be implemented via copy-and-`swap`, but the strong guarantee is not practical for all functions.
- A function can usually offer a guarantee no stronger than the weakest guarantee of the functions it calls.

Item 30: Understand the ins and outs of inlining.

Inline functions — what a *wonderful* idea! They look like functions, they act like functions, they're ever so much better than macros (see Item 2), and you can call them without having to incur the overhead of a function call. What more could you ask for?

You actually get more than you might think, because avoiding the cost of a function call is only part of the story. Compiler optimizations are typically designed for stretches of code that lack function calls, so when you inline a function, you may enable compilers to perform context-specific optimizations on the body of the function. Most compilers never perform such optimizations on “outlined” function calls.

In programming, however, as in life, there is no free lunch, and inline functions are no exception. The idea behind an inline function is to replace each call of that function with its code body, and it doesn't take a Ph.D. in statistics to see that this is likely to increase the size of your object code. On machines with limited memory, overzealous inlining can give rise to programs that are too big for the available space. Even with virtual memory, inline-induced code bloat can lead to additional paging, a reduced instruction cache hit rate, and the performance penalties that accompany these things.

On the other hand, if an inline function body is *very* short, the code generated for the function body may be smaller than the code generated for a function call. If that is the case, inlining the function may actually lead to *smaller* object code and a higher instruction cache hit rate!

Bear in mind that `inline` is a *request* to compilers, not a command. The request can be given implicitly or explicitly. The implicit way is to define a function inside a class definition:

Click here to view code image

```
class Person {
public:
  ...
  int age() const { return theAge; }    // an implicit inline
request: age is
  ...                                   // defined in a class
definition

private:
  int theAge;
};
```

Such functions are usually member functions, but Item 46 explains that friend functions can also be defined inside classes. When they are, they're also implicitly declared inline.

The explicit way to declare an inline function is to precede its definition with the `inline` keyword. For example, this is how the standard `max` template (from `<algorithm>`) is often implemented:

Click here to view code image

```
template<typename T>                               // an
explicit inline
inline const T& std::max(const T& a, const T& b)   //
request: std::max is
{ return a < b ? b : a; }                          //
preceded by "inline"
```

The fact that `max` is a template brings up the observation that both inline functions and templates are typically defined in header files. This leads some programmers to conclude that function templates must be inline. This conclusion is both invalid and potentially harmful, so it's worth looking into it a bit.

Inline functions must typically be in header files, because most build environments do inlining during compilation. In order to replace a function call with the body of the called function, compilers must know what the function looks like. (Some build environments can inline during linking, and a few — e.g., managed environments based on the .NET Common Language Infrastructure (CLI) — can actually inline at runtime. Such environments are the exception, however, not the rule. Inlining in most C++ programs is a compile-time activity.)

Templates are typically in header files, because compilers need to know what a template looks like in order to instantiate it when it's used. (Again, this is not universal. Some build environments perform template instantiation during linking. However, compile-time instantiation is more common.)

Template instantiation is independent of inlining. If you're writing a template and you believe that all the functions instantiated from the template should be inlined, declare the template `inline`; that's what's done with the `std::max` implementation above. But if you're writing a template for functions that you have no reason to want inlined, avoid declaring the template inline (either explicitly or implicitly). Inlining has costs, and you don't want to incur them without forethought. We've already mentioned how inlining can cause code bloat (a particularly important consideration for template authors — see Item 44), but there are other costs, too, which we'll discuss in a moment.

Before we do that, let's finish the observation that `inline` is a request that compilers may ignore. Most compilers refuse to inline functions they deem too complicated (e.g., those that contain loops or are recursive), and all but the most trivial calls to virtual functions defy inlining. This latter observation shouldn't be a surprise. `virtual` means “wait until runtime to figure out which function to call,” and `inline` means “before execution, replace the call site with the called function.” If compilers don't know which function will be called, you can hardly blame them for refusing to inline the function's body.

It all adds up to this: whether a given inline function is actually inlined depends on the build environment you're using — primarily on the compiler. Fortunately, most compilers have a diagnostic level that will result in a warning (see Item 53) if they fail to inline a function you've asked them to.

Sometimes compilers generate a function body for an inline function even when they are perfectly willing to inline the function. For example, if your program takes the address of an inline function, compilers must typically generate an outlined function body for it. How can they come up with a pointer to a function that doesn't exist? Coupled with the fact that compilers typically don't perform inlining across calls through function pointers, this means that calls to an inline function may or may not be inlined, depending on how the calls are made:

Click here to view code image

```
inline void f() {...}      // assume compilers are willing to
inline calls to f

void (*pf)() = f;          // pf points to f

...

f();                      // this call will be inlined,
because it's a "normal" call

pf();                     // this call probably won't be,
because it's through
                          // a function pointer
```

The specter of un-inlined inline functions can haunt you even if you never use function pointers, because programmers aren't necessarily the only ones asking for pointers to functions. Sometimes compilers generate out-of-line copies of constructors and destructors so that they can get pointers to those functions for use during construction and destruction of objects in arrays.

In fact, constructors and destructors are often worse candidates for inlining than a casual examination would indicate. For example, consider the constructor for class `Derived` below:

Click here to view code image

```
class Base {
public:
...

private:
   std::string bm1, bm2;              // base members 1 and
2
};
```

```
class Derived: public Base {
public:
  Derived() {}                            // Derived's ctor is
empty — or is it?
  ...

private:
  std::string dm1, dm2, dm3;              // derived members 1–3
};
```

This constructor looks like an excellent candidate for inlining, since it contains no code. But looks can be deceiving.

C++ makes various guarantees about things that happen when objects are created and destroyed. When you use `new`, for example, your dynamically created objects are automatically initialized by their constructors, and when you use `delete`, the corresponding destructors are invoked. When you create an object, each base class of and each data member in that object is automatically constructed, and the reverse process regarding destruction automatically occurs when an object is destroyed. If an exception is thrown during construction of an object, any parts of the object that have already been fully constructed are automatically destroyed. In all these scenarios, C++ says *what* must happen, but it doesn't say *how*. That's up to compiler implementers, but it should be clear that those things don't happen by themselves. There has to be some code in your program to make those things happen, and that code — the code written by compilers and inserted into your program during compilation — has to go somewhere. Sometimes it ends up in constructors and destructors, so we can imagine implementations generating code equivalent to the following for the allegedly empty `Derived` constructor above:

Click here to view code image

```
Derived::Derived()                          // conceptual
implementation of
{                                           // "empty" Derived
ctor

Base::Base();                              // initialize Base
part

try { dm1.std::string::string(); }         // try to construct
dm1
catch (...) {                           // if it throws,
   Base::~Base();                        // destroy base
class part and
   throw;                                // propagate the
exception
}

try { dm2.std::string::string(); }      // try to construct
dm2
catch(...) {                            // if it throws,
   dm1.std::string::~string();           // destroy dm1,
   Base::~Base();                        // destroy base
class part, and
   throw;                                // propagate the
exception
}

try { dm3.std::string::string(); }      // construct dm3
catch(...) {                            // if it throws,
   dm2.std::string::~string();           // destroy dm2,
   dm1.std::string::~string();           // destroy dm1,
   Base::~Base();                        // destroy base
class part, and
   throw;                                // propagate the
exception
}
}
```

This code is unrepresentative of what real compilers emit, because real compilers deal with exceptions in more sophisticated ways. Still, this accurately reflects the behavior that `Derived`'s “empty” constructor must offer. No matter how sophisticated a compiler's exception implementation, `Derived`'s constructor must at least call constructors for its data members and base class, and those calls (which might themselves be inlined) could affect its attractiveness for inlining.

The same reasoning applies to the `Base` constructor, so if it's inlined, all the code inserted into it is also inserted into the `Derived` constructor (via the `Derived` constructor's call to the `Base` constructor). And if the `string` constructor also happens to be inlined, the `Derived` constructor will gain *five copies* of that function's code, one for each of the five strings in a `Derived` object (the two it inherits plus the three it declares itself). Perhaps now it's clear why it's not a no-brain decision whether to inline `Derived`'s constructor. Similar considerations apply to `Derived`'s destructor, which, one way or another, must see to it that all the objects initialized by `Derived`'s constructor are properly destroyed.

Library designers must evaluate the impact of declaring functions `inline`, because it's impossible to provide binary upgrades to the client visible inline functions in a library. In other words, if `f` is an inline function in a library, clients of the library compile the body of `f` into their applications. If a library implementer later decides to change `f`, all clients who've used `f` must recompile. This is often undesirable. On the other hand, if `f` is a non-inline function, a modification to `f` requires only that clients relink. This is a substantially less onerous burden than recompiling and, if the library containing the function is dynamically linked, one that may be absorbed in a way that's completely transparent to clients.

For purposes of program development, it is important to keep all these considerations in mind, but from a practical point of view during coding, one fact dominates all others: most debuggers have trouble with inline functions. This should be no great revelation. How do you set a breakpoint in a function that isn't there? Although some build environments manage to support debugging of inlined functions, many environments simply disable inlining for debug builds.

This leads to a logical strategy for determining which functions should be declared inline and which should not. Initially, don't inline anything, or at least limit your inlining to those functions that must be inline (see Item 46) or are truly trivial (such as `Person::age` on page 135). By employing inlines cautiously, you facilitate your use of a debugger, but you also put inlining in its proper place: as a hand-applied optimization. Don't forget the empirically determined rule of 80-20, which states that a typical program spends 80% of its time executing only 20% of its code. It's an important rule, because it reminds you that your goal as a software developer is to identify the 20% of your code that can increase your program's overall performance. You can inline and otherwise tweak your functions until the cows come home, but it's wasted effort unless you're focusing on the *right* functions.

Things to Remember

- Limit most inlining to small, frequently called functions. This facilitates debugging and binary upgradability, minimizes potential code bloat, and maximizes the chances of greater program speed.

- Don't declare function templates `inline` just because they appear in header files.

Item 31: Minimize compilation dependencies between files

So you go into your C++ program and make a minor change to the implementation of a class. Not the class interface, mind you, just the implementation; only the private stuff. Then you rebuild the program, figuring that the exercise should take only a few seconds. After all, only one class has been modified. You click on Build or type `make` (or some equivalent), and you are astonished, then mortified, as you realize that the whole *world* is being recompiled and relinked! Don't you just *hate* it when that happens?

The problem is that C++ doesn't do a very good job of separating interfaces from implementations. A class definition specifies not only a class interface but also a fair number of implementation details. For example:

Click here to view code image

```
class Person {
public:
  Person(const std::string& name, const Date& birthday,
         const Address& addr);
  std::string name() const;
  std::string birthDate() const;
  std::string address() const;
  ...

private:
      std::string theName;        // implementation detail
      Date theBirthDate;          // implementation detail
      Address theAddress;         // implementation detail
};
```

Here, class `Person` can't be compiled without access to definitions for the classes the `Person` implementation uses, namely, `string`, `Date`, and `Address`. Such definitions are typically provided through `#include` directives, so in the file defining the `Person` class, you are likely to find something like this:

```
#include <string>
#include "date.h"
#include "address.h"
```

Unfortunately, this sets up a compilation dependency between the file defining `Person` and these header files. If any of these header files is changed, or if any of the header files *they* depend on changes, the file containing the `Person` class must be recompiled, as must any files that use `Person`. Such cascading compilation dependencies have caused many a project untold grief.

You might wonder why C++ insists on putting the implementation details of a class in the class definition. For example, why can't you define `Person` this way, specifying the implementation details of the class separately?

Click here to view code image

```
namespace std {
     class string;             // forward declaration (an
incorrect
}                              // one — see below)

class Date;                    // forward declaration
class Address;                 // forward declaration

class Person {
public:
      Person(const std::string& name, const Date& birthday,
                 const Address& addr);
      std::string name() const;
      std::string birthDate() const;
      std::string address() const;
    ...
};
```

If that were possible, clients of `Person` would have to recompile only if the interface to the class changed.

There are two problems with this idea. First, `string` is not a class, it's a typedef (for `basic_string<char>`). As a result, the forward declaration for `string` is incorrect. The proper forward declaration is substantially more complex, because it involves additional templates. That doesn't matter, however, because you shouldn't try to manually declare parts of the standard library. Instead, simply use the proper `#include`s and be done with it. Standard headers are unlikely to be a compilation bottleneck, especially if your build environment allows you to take advantage of precompiled headers. If parsing standard headers really is a problem, you may need to change your interface design to avoid using the parts of the standard library that give rise to the undesirable `#include`s.

The second (and more significant) difficulty with forward-declaring everything has to do with the need for compilers to know the size of objects during compilation. Consider:

Click here to view code image

```
int main()
{
int x;                    // define an int

Person p( params );       // define a Person
   ...
}
```

When compilers see the definition for `x`, they know they must allocate enough space (typically on the stack) to hold an `int`. No problem. Each compiler knows how big an `int` is. When compilers see the definition for `p`, they know they have to allocate enough space for a `Person`, but how are they supposed to know how big a `Person` object is? The only way they can get that information is to consult the class definition, but if it were legal for a class definition to omit the implementation details, how would compilers know how much space to allocate?

This question fails to arise in languages like Smalltalk and Java, because, when an object is defined in such languages, compilers allocate only enough space for a *pointer* to an object. That is, they handle the code above as if it had been written like this:

Click here to view code image

```
int main()
{
  int x;                    // define an int
  Person *p;            // define a pointer to a Person
  ...
}
```

This, of course, is legal C++, so you can play the “hide the object implementation behind a pointer” game yourself. One way to do that for `Person` is to separate it into two classes, one offering only an interface, the other implementing that interface. If the implementation class is named `PersonImpl`, `Person` would be defined like this:

Click here to view code image

```
#include <string>                      // standard library
components
                                       // shouldn't be
forward-declared

#include <memory>                      // for
tr1::shared_ptr; see below

class PersonImpl;                      // forward decl of
Person impl. class
class Date;                            // forward decls of
classes used in

class Address;                         // Person interface
class Person {
public:
Person(const std::string& name, const Date& birthday,
        const Address& addr);
std::string name() const;
std::string birthDate() const;
std::string address() const;
...

private:                                   // ptr to
implementation;
  std::tr1::shared_ptr<PersonImpl> pImpl;  // see Item 13 for
info on
};                                         //
std::tr1::shared_ptr
```

Here, the main class (`Person`) contains as a data member nothing but a pointer (here, a `tr1::shared_ptr` — see Item 13) to its implementation class (`PersonImpl`). Such a design is often said to be using the *pimpl idiom* (“pointer to implementation”). Within such classes, the name of the pointer is often `pImpl`, as it is above.

With this design, clients of `Person` are divorced from the details of dates, addresses, and persons. The implementations of those classes can be modified at will, but `Person` clients need not recompile. In addition, because they're unable to see the details of `Person`'s implementation, clients are unlikely to write code that somehow depends on those details. This is a true separation of interface and implementation.

The key to this separation is replacement of dependencies on *definitions* with dependencies on *declarations*. That's the essence of minimizing compilation dependencies: make your header files self-sufficient whenever it's practical, and when it's not, depend on declarations in other files, not definitions. Everything else flows from this simple design strategy. Hence:

- **Avoid using objects when object references and pointers will do.** You may define references and pointers to a type with only a declaration for the type. Defining *objects* of a type necessitates the presence of the type's definition.
- **Depend on class declarations instead of class definitions whenever you can.** Note that you *never* need a class definition to declare a function using that class, not even if the function passes or returns the class type by value:

Click here to view code image

```
class Date;                          // class declaration

Date today();                        // fine — no definition
void clearAppointments(Date d);      // of Date is needed
```

Of course, pass-by-value is generally a bad idea (see Item 20), but if you find yourself using it for some reason, there's still no justification for introducing unnecessary compilation dependencies.

The ability to declare `today` and `clearAppointments` without defining `Date` may surprise you, but it's not as curious as it seems. If anybody *calls* those functions, `Date`'s definition must have been seen prior to the call. Why bother to declare functions that nobody calls, you wonder? Simple. It's not that *nobody* calls them, it's that *not everybody* calls them. If you have a library containing dozens of function declarations, it's unlikely that every client calls every function. By moving the onus of providing class definitions from your header file of function *declarations* to clients' files containing function *calls*, you eliminate artificial client dependencies on type definitions they don't really need.

- **Provide separate header files for declarations and definitions.** In order to facilitate adherence to the above guidelines, header files need to come in pairs: one for declarations, the other for definitions. These files must be kept consistent, of course. If a declaration is changed in one place, it must be changed in both. As a result, library clients should always `#include` a declaration file instead of forward-declaring something themselves, and library authors should provide both header files. For example, the `Date` client wishing to declare `today` and `clearAppointments` shouldn't manually forward-declare `Date` as shown above. Rather, it should `#include` the appropriate header of declarations:

Click here to view code image

```
#include "datefwd.h"            // header file declaring (but
not
                                // defining) class Date

Date today();                   // as before
void clearAppointments(Date d);
```

The name of the declaration-only header file “`datefwd.h`” is based on the header `<iosfwd>` from the standard C++ library (see Item 54). `<iosfwd>` contains declarations of iostream components whose corresponding definitions are in several different headers, including `<sstream>`, `<streambuf>`, `<fstream>`, and `<iostream>`.

`<iosfwd>` is instructive for another reason, and that's to make clear that the advice in this Item applies as well to templates as to non-templates. Although Item 30 explains that in many build environments, template definitions are typically found in header files, some build environments allow template definitions to be in non-header files, so it still makes sense to provide declaration-only headers for templates. `<iosfwd>` is one such header.

C++ also offers the `export` keyword to allow the separation of template declarations from template definitions. Unfortunately, compiler support for `export` is scanty, and real-world experience with `export` is scantier still. As a result, it's too early to say what role `export` will play in effective C++ programming.

Classes like `Person` that employ the pimpl idiom are often called *Handle classes*. Lest you wonder how such classes actually do anything, one way is to forward all their function calls to the corresponding implementation classes and have those classes do the real work. For example, here's how two of `Person`'s member functions could be implemented:

Click here to view code image

```
   #include "Person.h"         // we're implementing the Person
class,
                             // so we must #include its class
definition


#include "PersonImpl.h"      // we must also #include
PersonImpl's class
                             // definition, otherwise we
couldn't call
                             // its member functions; note
that
                             // PersonImpl has exactly the
same public
                             // member functions as Person —
their
                             // interfaces are identical

Person::Person(const std::string& name, const Date& birthday,
               const Address& addr)
: pImpl(new PersonImpl(name, birthday, addr))
{}

std::string Person::name() const
{
  return pImpl->name();
}
```

Note how the `Person` constructor calls the `PersonImpl` constructor (by using `new` — see Item 16) and how `Person::name` calls `PersonImpl::name`. This is important.

Making `Person` a Handle class doesn't change what `Person` does, it just changes the way it does it.

An alternative to the Handle class approach is to make `Person` a special kind of abstract base class called an *Interface class*. The purpose of such a class is to specify an interface for derived classes (see Item 34). As a result, it typically has no data members, no constructors, a virtual destructor (see Item 7), and a set of pure virtual functions that specify the interface.

Interface classes are akin to Java's and .NET's Interfaces, but C++ doesn't impose the restrictions on Interface classes that Java and .NET impose on Interfaces. Neither Java nor .NET allow data members or function implementations in Interfaces, for example, but C++ forbids neither of these things. C++'s greater flexibility can be useful. As Item 36 explains, the implementation of non-virtual functions should be the same for all classes in a hierarchy, so it makes sense to implement such functions as part of the Interface class that declares them.

An Interface class for `Person` could look like this:

Click here to view code image

```
class Person {
public:
  virtual ~Person();

  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
  virtual std::string address() const = 0;
  ...
};
```

Clients of this class must program in terms of `Person` pointers and references, because it's not possible to instantiate classes containing pure virtual functions. (It is, however, possible to instantiate classes *derived* from `Person` — see below.) Like clients of Handle classes, clients of Interface classes need not recompile unless the Interface class's interface is modified.

Clients of an Interface class must have a way to create new objects. They typically do it by calling a function that plays the role of the constructor for the derived classes that are actually instantiated. Such functions are typically called factory functions (see Item 13) or *virtual constructors*. They return pointers (preferably smart pointers — see Item 18) to dynamically allocated objects that support the Interface class's interface. Such functions are often declared `static` inside the Interface class:

Click here to view code image

```
class Person {
public:
...

static std::tr1::shared_ptr<Person>    // return a
tr1::shared_ptr to a new
   create(const std::string& name,      // Person initialized
with the
          const Date& birthday,         // given params; see
Item 18 for
          const Address& addr);         // why a
tr1::shared_ptr is returned
...
};
```

Clients use them like this:

Click here to view code image

```
std::string name;
Date dateOfBirth;
Address address;
...

// create an object supporting the Person interface
std::tr1::shared_ptr<Person> pp(Person::create(name,
dateOfBirth, address));

...

std::cout << pp->name()                 // use the object via
the
          << " was born on "            // Person interface
    << pp->birthDate()
          << " and now lives at "
          << pp->address();
...                                     // the object is
automatically
                                        // deleted when pp
goes out of
```

```
                                                // scope — see Item
13
```

At some point, of course, concrete classes supporting the Interface class's interface must be defined and real constructors must be called. That all happens behind the scenes inside the files containing the implementations of the virtual constructors. For example, the Interface class `Person` might have a concrete derived class `RealPerson` that provides implementations for the virtual functions it inherits:

Click here to view code image

```
class RealPerson: public Person {
public:
  RealPerson(const std::string& name, const Date& birthday,
             const Address& addr)
  : theName(name), theBirthDate(birthday), theAddress(addr)
  {}

  virtual ~RealPerson() {}

  std::string name() const;        // implementations of
these
  std::string birthDate() const;   // functions are not
shown, but
  std::string address() const;     // they are easy to
imagine

private:
  std::string theName;
  Date theBirthDate;
  Address theAddress;
};
```

Given `RealPerson`, it is truly trivial to write `Person::create`:

Click here to view code image

```
std::tr1::shared_ptr<Person> Person::create(const
std::string& name,
                                            const Date&
birthday,
                                            const Address&
addr)
```

```
{
  return std::tr1::shared_ptr<Person>(new RealPerson(name,
birthday, addr));
}
```

A more realistic implementation of `Person::create` would create different types of derived class objects, depending on e.g., the values of additional function parameters, data read from a file or database, environment variables, etc.

`RealPerson` demonstrates one of the two most common mechanisms for implementing an Interface class: it inherits its interface specification from the Interface class (`Person`), then it implements the functions in the interface. A second way to implement an Interface class involves multiple inheritance, a topic explored in Item 40.

Handle classes and Interface classes decouple interfaces from implementations, thereby reducing compilation dependencies between files. Cynic that you are, I know you're waiting for the fine print. “What does all this hocus-pocus cost me?” you mutter. The answer is the usual one in computer science: it costs you some speed at runtime, plus some additional memory per object.

In the case of Handle classes, member functions have to go through the implementation pointer to get to the object's data. That adds one level of indirection per access. And you must add the size of this implementation pointer to the amount of memory required to store each object. Finally, the implementation pointer has to be initialized (in the Handle class's constructors) to point to a dynamically allocated implementation object, so you incur the overhead inherent in dynamic memory allocation (and subsequent deallocation) and the possibility of encountering `bad_alloc` (out-of-memory) exceptions.

For Interface classes, every function call is virtual, so you pay the cost of an indirect jump each time you make a function call (see Item 7). Also, objects derived from the Interface class must contain a virtual table pointer (again, see Item 7). This pointer may increase the amount of memory needed to store an object, depending on whether the Interface class is the exclusive source of virtual functions for the object.

Finally, neither Handle classes nor Interface classes can get much use out of inline functions. Item 30 explains why function bodies must typically be in header files in order to be inlined, but Handle and Interface classes are specifically designed to hide implementation details like function bodies.

It would be a serious mistake, however, to dismiss Handle classes and Interface classes simply because they have a cost associated with them. So do virtual functions, and you wouldn't want to forgo those, would you? (If so, you're reading the wrong book.) Instead, consider using these techniques in an evolutionary manner. Use Handle classes and Interface classes during development to minimize the impact on clients when implementations change. Replace Handle classes and Interface classes with concrete classes for production use when it can be shown that the difference in speed and/or size is significant enough to justify the increased coupling between classes.

Things to Remember

- The general idea behind minimizing compilation dependencies is to depend on declarations instead of definitions. Two approaches based on this idea are Handle classes and Interface classes.
- Library header files should exist in full and declaration-only forms. This applies regardless of whether templates are involved.

6. Inheritance and Object-Oriented Design

Object-oriented programming (OOP) has been the rage for almost two decades, so it's likely that you have some experience with the ideas of inheritance, derivation, and virtual functions. Even if you've been programming only in C, you've surely not escaped the OOP hoopla.

Still, OOP in C++ is probably a bit different from what you're used to. Inheritance can be single or multiple, and each inheritance link can be public, protected, or private. Each link can also be virtual or non-virtual. Then there are the member function options. Virtual? Non-virtual? Pure virtual? And the interactions with other language features. How do default parameter values interact with virtual functions? How does inheritance affect C++'s name lookup rules? And what about design options? If a class's behavior needs to be modifiable, is a virtual function the best way to do that?

This chapter sorts it all out. Furthermore, I explain what the different features in C++ really *mean* — what you are really *expressing* when you use a particular construct. For example, public inheritance means “is-a,” and if you try to make it mean anything else, you'll run into trouble. Similarly, a virtual function means “interface must be inherited,” while a non-virtual function means “both interface and implementation must be inherited.” Failing to distinguish between these meanings has caused C++ programmers considerable grief.

If you understand the meanings of C++'s various features, you'll find that your outlook on OOP changes. Instead of it being an exercise in differentiating between language features, it will become a matter of determining what you want to say about your software system. And once you know what you want to say, the translation into C++ is not terribly demanding.

Item 32: Make sure public inheritance models “is-a.”

In his book, *Some Must Watch While Some Must Sleep* (W. H. Freeman and Company, 1974), William Dement relates the story of his attempt to fix in the minds of his students the most important lessons of his course. It is claimed, he told his class, that the average British school-child remembers little more history than that the Battle of Hastings was in 1066. If a child remembers little else, Dement emphasized, he or she remembers the date 1066. For the students in *his* course, Dement went on, there were only a few central messages, including, interestingly enough, the fact that sleeping pills cause insomnia. He implored his students to remember these few critical facts even if they forgot everything else discussed in the course, and he returned to these fundamental precepts repeatedly during the term.

At the end of the course, the last question on the final exam was, “Write one thing from the course that you will surely remember for the rest of your life.” When Dement graded the exams, he was stunned. Nearly everyone had written “1066.”

It is thus with great trepidation that I proclaim to you now that the single most important rule in object-oriented programming with C++ is this: public inheritance means “is-a.” Commit this rule to memory.

If you write that class `D` (“Derived”) publicly inherits from class `B` (“Base”), you are telling C++ compilers (as well as human readers of your code) that every object of type `D` is also an object of type `B`, but *not vice versa*. You are saying that `B` represents a more general concept than `D`, that `D` represents a more specialized concept than `B`. You are asserting that anywhere an object of type `B` can be used, an object of type `D` can be used just as well, because every object of type `D` *is* an object of type `B`. On the other hand, if you need an object of type `D`, an object of type `B` will not do: every `D` is-a `B`, but not vice versa.

C++ enforces this interpretation of public inheritance. Consider this example:

```
class Person {...};

class Student: public Person {...};
```

We know from everyday experience that every student is a person, but not every person is a student. That is exactly what this hierarchy asserts. We expect that anything that is true of a person — for example, that he or she has a date of birth — is also true of a student. We do not expect that everything that is true of a student — that he or she is enrolled in a particular school, for instance — is true of people in general. The notion of a person is more general than is that of a student; a student is a specialized type of person.

Within the realm of C++, any function that expects an argument of type `Person` (or pointer-to-`Person` or reference-to-`Person`) will also take a `Student` object (or pointer-to-`Student` or reference-to-`Student`):

Click here to view code image

```
void eat(const Person& p);            // anyone can eat

void study(const Student& s);         // only students study

Person p;                             // p is a Person
Student s;                            // s is a Student

eat(p);                               // fine, p is a Person

eat(s);                               // fine, s is a
Student,
                                      // and a Student is-a

Person

study(s);                             // fine

study(p);                             // error! p isn't a
Student
```

This is true only for *public* inheritance. C++ will behave as I've described only if `Student` is publicly derived from `Person`. Private inheritance means something entirely different (see Item 39), and protected inheritance is something whose meaning eludes me to this day.

The equivalence of public inheritance and is-a sounds simple, but sometimes your intuition can mislead you. For example, it is a fact that a penguin is a bird, and it is a fact that birds can fly. If we naively try to express this in C++, our effort yields:

Click here to view code image

```
class Bird {
public:
  virtual void fly();                  // birds can fly

  ...
};
```

```
class Penguin: public Bird {            // penguins are birds
  ...
};
```

Suddenly we are in trouble, because this hierarchy says that penguins can fly, which we know is not true. What happened?

In this case, we are the victims of an imprecise language: English. When we say that birds can fly, we don't mean that *all* types of birds can fly, only that, in general, birds have the ability to fly. If we were more precise, we'd recognize that there are several types of non-flying birds, and we would come up with the following hierarchy, which models reality much better:

Click here to view code image

```
class Bird {
  ...                                         // no fly
function is declared
};

class FlyingBird: public Bird {
public:
  virtual void fly();
  ...
};

class Penguin: public Bird {

  ...                                         // no fly
function is declared

};
```

This hierarchy is much more faithful to what we really know than was the original design.

Yet we're not finished with these fowl matters, because for some software systems, there may be no need to distinguish between flying and non-flying birds. If your application has much to do with beaks and wings and nothing to do with flying, the original two-class hierarchy might be quite satisfactory. That's a simple reflection of the fact that there is no one ideal design for all software. The best design depends on what the system is expected to do, both now and in the future. If your application has no knowledge of flying and isn't expected to ever have any, failing to distinguish between flying and non-flying birds may be a perfectly valid design decision. In fact, it may be preferable to a design that does distinguish between them, because such a distinction would be absent from the world you are trying to model.

There is another school of thought on how to handle what I call the “All birds can fly, penguins are birds, penguins can't fly, uh oh” problem. That is to redefine the `fly` function for penguins so that it generates a runtime error:

Click here to view code image

```
void error(const std::string& msg);       // defined
elsewhere

class Penguin: public Bird {
public:
  virtual void fly() { error("Attempt to make a penguin
fly!"); }

  ...

};
```

It's important to recognize that this says something different from what you might think. This does *not* say, “Penguins can't fly.” This says, “Penguins can fly, but it's an error for them to actually try to do it.”

How can you tell the difference? From the time at which the error is detected. The injunction, “Penguins can't fly,” can be enforced by compilers, but violations of the rule, “It's an error for penguins to actually try to fly,” can be detected only at runtime.

To express the constraint, “Penguins can't fly — *period*,” you make sure that no such function is defined for `Penguin` objects:

Click here to view code image

```
class Bird {

  ...                              // no fly function is
declared

};

class Penguin: public Bird {
  ...                                    // no fly function is
declared

};
```

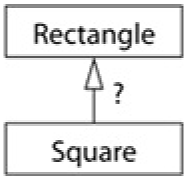
If you now try to make a penguin fly, compilers will reprimand you for your transgression:

```
Penguin p;

p.fly();                             // error!
```

This is very different from the behavior you get if you adopt the approach that generates runtime errors. With that methodology, compilers won't say a word about the call to `p.fly`. Item 18 explains that good interfaces prevent invalid code from compiling, so you should prefer the design that rejects penguin flight attempts during compilation to the one that detects them only at runtime.

Perhaps you'll concede that your ornithological intuition may be lacking, but you can rely on your mastery of elementary geometry, right? I mean, how complicated can rectangles and squares be?

Well, answer this simple question: should class `Square` publicly inherit from class `Rectangle`?



“Duh!” you say, “Of course it should! Everybody knows that a square is a rectangle, but generally not vice versa.” True enough, at least in school. But I don't think we're in school anymore.

Consider this code:

Click here to view code image

```
class Rectangle {
public:
  virtual void setHeight(int newHeight);
  virtual void setWidth(int newWidth);

  virtual int height() const;               // return current
values
  virtual int width() const;

  ...

};

void makeBigger(Rectangle& r)               // function to
increase r's area
{
  int oldHeight = r.height();

  r.setWidth(r.width() + 10);               // add 10 to r's
width

  assert(r.height() == oldHeight);          // assert that
r's
}                                           // height is
unchanged
```

Clearly, the assertion should never fail. `makeBigger` only changes `r`'s width. Its height is never modified.

Now consider this code, which uses public inheritance to allow squares to be treated like rectangles:

Click here to view code image

```
class Square: public Rectangle {...};

Square s;

...

assert(s.width() == s.height());           // this must be
true for all squares

makeBigger(s);                             // by inheritance,
s is-a Rectangle,
                                           // so we can
increase its area
```

```
assert(s.width() == s.height());           // this must still
be true
                                           // for all squares
```

It's just as clear that this second assertion should also never fail. By definition, the width of a square is the same as its height.

But now we have a problem. How can we reconcile the following assertions?

- Before calling `makeBigger`, `s`'s height is the same as its width;
- Inside `makeBigger`, `s`'s width is changed, but its height is not;
- After returning from `makeBigger`, `s`'s height is again the same as its width. (Note that `s` is passed to `makeBigger` by reference, so `makeBigger` modifies `s` itself, not a copy of `s`.)

Well?

Welcome to the wonderful world of public inheritance, where the instincts you've developed in other fields of study — including mathematics — may not serve you as well as you expect. The fundamental difficulty in this case is that something applicable to a rectangle (its width may be modified independently of its height) is not applicable to a square (its width and height must be the same). But public inheritance asserts that everything that applies to base class objects — *everything!* — also applies to derived class objects. In the case of rectangles and squares (as well as an example involving sets and lists in Item 38), that assertion fails to hold, so using public inheritance to model their relationship is simply incorrect. Compilers will let you do it, but as we've just seen, that's no guarantee the code will behave properly. As every programmer must learn (some more often than others), just because the code compiles doesn't mean it will work.

Don't fret that the software intuition you've developed over the years will fail you as you approach object-oriented design. That knowledge is still valuable, but now that you've added inheritance to your arsenal of design alternatives, you'll have to augment your intuition with new insights to guide you in inheritance's proper application. In time, the notion of having `Penguin` inherit from `Bird` or `Square` inherit from `Rectangle` will give you the same funny feeling you probably get now when somebody shows you a function several pages long. It's *possibly* the right way to approach things, it's just not very likely.

The is-a relationship is not the only one that can exist between classes. Two other common inter-class relationships are “has-a” and “is-implemented-in-terms-of.” These relationships are considered in Items 38 and 39. It's not uncommon for C++ designs to go awry because one of these other important relationships was incorrectly modeled as is-a, so you should make sure that you understand the differences among these relationships and that you know how each is best modeled in C++.

Things to Remember

- Public inheritance means “is-a.” Everything that applies to base classes must also apply to derived classes, because every derived class object *is* a base class object.

Item 33: Avoid hiding inherited names

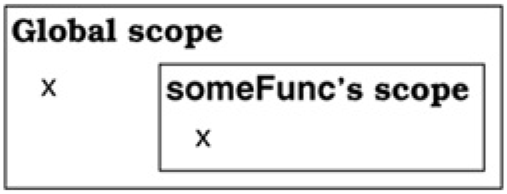
Shakespeare had a thing about names. “What's in a name?” he asked, “A rose by any other name would smell as sweet.” The Bard also wrote, “he that filches from me my good name ... makes me poor indeed.” Right. Which brings us to inherited names in C++.

The matter actually has nothing to do with inheritance. It has to do with scopes. We all know that in code like this,

Click here to view code image

```
int x;                        // global variable

void someFunc()
{
  double x;                   // local variable

  std::cin >> x;              // read a new value for local x
}
```

the statement reading into `x` refers to the local variable `x` instead of the global variable `x`, because names in inner scopes hide (“shadow”) names in outer scopes. We can visualize the scope situation this way:

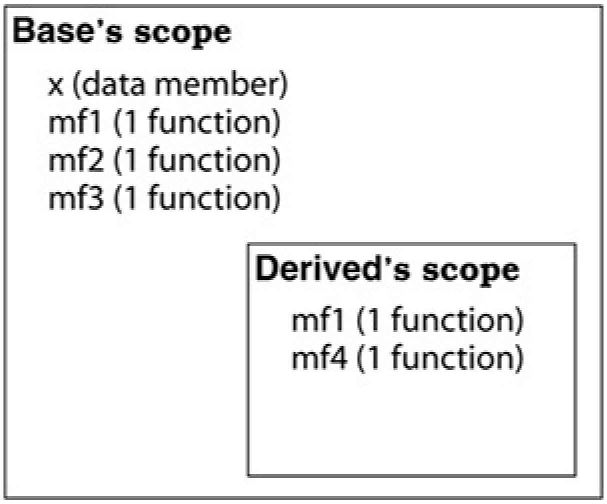


When compilers are in `someFunc`'s scope and they encounter the name `x`, they look in the local scope to see if there is something with that name. Because there is, they never examine any other scope. In this case, `someFunc`'s `x` is of type `double` and the global `x` is of type `int`, but that doesn't matter. C++'s name-hiding rules do just that: hide *names*. Whether the names correspond to the same or different types is immaterial. In this case, a `double` named `x` hides an `int` named `x`.

Enter inheritance. We know that when we're inside a derived class member function and we refer to something in a base class (e.g., a member function, a typedef, or a data member), compilers can find what we're referring to because derived classes inherit the things declared in base classes. The way that actually works is that the scope of a derived class is nested inside its base class's scope. For example:

Click here to view code image

```
class Base {
private:
  int x;

public:
  virtual void mf1() = 0;
  virtual void mf2();
  void mf3();

  ...
};

class Derived: public Base {
public:
  virtual void mf1();
  void mf4();

  ...
};
```



This example includes a mix of public and private names as well as names of both data members and member functions. The member functions are pure virtual, simple (impure) virtual, and non-virtual. That's to emphasize that we're talking about *names*. The example could also have included names of types, e.g., enums, nested classes, and typedefs. The only thing that matters in this discussion is that they're names. What they're names *of* is irrelevant. The example uses single inheritance, but once you understand what's happening under single inheritance, C++'s behavior under multiple inheritance is easy to anticipate.

Suppose `mf4` in the derived class is implemented, in part, like this:

```
void Derived::mf4()
{

  ...
  mf2();

  ...
}
```

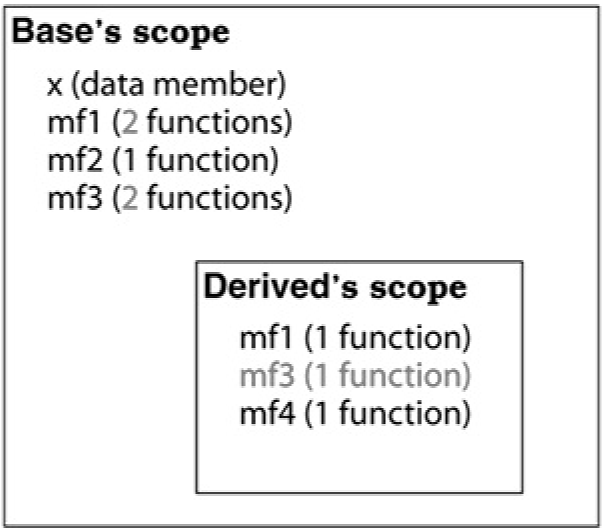
When compilers see the use of the name `mf2` here, they have to figure out what it refers to. They do that by searching scopes for a declaration of something named `mf2`. First they look in the local scope (that of `mf4`), but they find no declaration for anything called `mf2`. They then search the containing scope, that of the class `Derived`. They still find nothing named `mf2`, so they move on to the next containing scope, that of the base class. There they find something named `mf2`, so the search stops. If there were no `mf2` in `Base`, the search would continue, first to the namespace(s) containing `Derived`, if any, and finally to the global scope.

The process I just described is accurate, but it's not a comprehensive description of how names are found in C++. Our goal isn't to know enough about name lookup to write a compiler, however. It's to know enough to avoid unpleasant surprises, and for that task, we already have plenty of information.

Consider the previous example again, except this time let's overload `mf1` and `mf3`, and let's add a version of `mf3` to `Derived`. (As Item 36 explains, `Derived`'s declaration of `mf3` — an inherited non-virtual function — makes this design instantly suspicious, but in the interest of understanding name visibility under inheritance, we'll overlook that.)

Click here to view code image

```
class Base {
private:
  int x;

public:
  virtual void mf1() = 0;
  virtual void mf1(int);

  virtual void mf2();

  void mf3();
  void mf3(double);
  ...
};

class Derived: public Base {
public:
  virtual void mf1();
  void mf3();
  void mf4();
  ...
};
```


Base's scope
x (data member)
mf1 (2 functions)
mf2 (1 function)
mf3 (2 functions)
Derived's scope
mf1 (1 function)
mf3 (1 function)
mf4 (1 function)

This code leads to behavior that surprises every C++ programmer the first time they encounter it. The scope-based name hiding rule hasn't changed, so *all* functions named `mf1` and `mf3` in the base class are hidden by the functions named `mf1` and `mf3` in the derived class. From the perspective of name lookup, `Base::mf1` and `Base::mf3` are no longer inherited by `Derived`!

Click here to view code image

```
Derived d;
int x;

...
d.mf1();                   // fine, calls Derived::mf1
d.mf1(x);                  // error! Derived::mf1 hides
Base::mf1

d.mf2();                   // fine, calls Base::mf2

d.mf3();                   // fine, calls Derived::mf3
d.mf3(x);                  // error! Derived::mf3 hides
Base::mf3
```

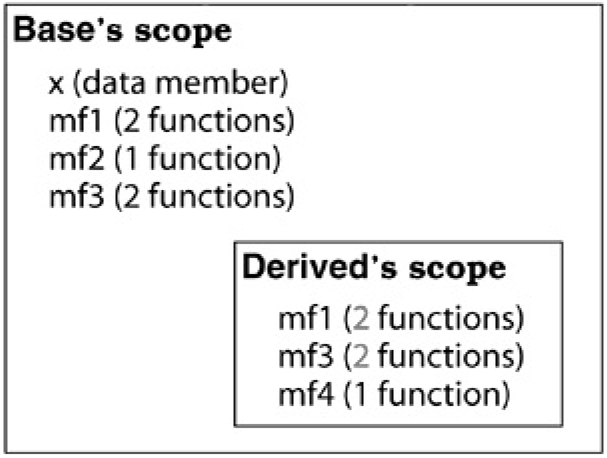
As you can see, this applies even though the functions in the base and derived classes take different parameter types, and it also applies regardless of whether the functions are virtual or non-virtual. In the same way that, at the beginning of this Item, the `double x` in the function `someFunc` hides the `int x` at global scope, here the function `mf3` in `Derived` hides a `Base` function named `mf3` that has a different type.

The rationale behind this behavior is that it prevents you from accidentally inheriting overloads from distant base classes when you create a new derived class in a library or application framework. Unfortunately, you typically *want* to inherit the overloads. In fact, if you're using public inheritance and you don't inherit the overloads, you're violating the is-a relationship between base and derived classes that Item 32 explains is fundamental to public inheritance. That being the case, you'll almost always want to override C++'s default hiding of inherited names.

You do it with *`using` declarations*:

Click here to view code image

```
class Base {
private:
  int x;

public:
  virtual void mf1() = 0;
  virtual void mf1(int);

  virtual void mf2();

  void mf3();
  void mf3(double);
  ...
};

class Derived: public Base {
public:
  using Base::mf1;       // make all things in Base named mf1
and mf3
  using Base::mf3;       // visible (and public) in Derived's
scope

  virtual void mf1();
  void mf3();
  void mf4();
  ...
};
```


Base's scope
x (data member)
mf1 (2 functions)
mf2 (1 function)
mf3 (2 functions)
Derived's scope
mf1 (2 functions)
mf3 (2 functions)
mf4 (1 function)

Now inheritance will work as expected:

```
Derived d;
int x;

...

d.mf1();                    // still fine, still calls
Derived::mf1
d.mf1(x);                   // now okay, calls Base::mf1

d.mf2();                    // still fine, still calls Base::mf2

d.mf3();                    // fine, calls Derived::mf3
d.mf3(x);                   // now okay, calls Base::mf3
```

This means that if you inherit from a base class with overloaded functions and you want to redefine or override only some of them, you need to include a `using` declaration for each name you'd otherwise be hiding. If you don't, some of the names you'd like to inherit will be hidden.

It's conceivable that you sometimes won't want to inherit all the functions from your base classes. Under public inheritance, this should never be the case, because, again, it violates public inheritance's is-a relationship between base and derived classes. (That's why the `using` declarations above are in the public part of the derived class: names that are public in a base class should also be public in a publicly derived class.) Under private inheritance (see Item 39), however, it can make sense. For example, suppose `Derived` privately inherits from `Base`, and the only version of `mf1` that `Derived` wants to inherit is the one taking no parameters. A `using` declaration won't do the trick here, because a `using` declaration makes *all* inherited functions with a given name visible in the derived class. No, this is a case for a different technique, namely, a simple forwarding function:

Click here to view code image

```
class Base {
public:
  virtual void mf1() = 0;
  virtual void mf1(int);

  ...                                    // as before
};

class Derived: private Base {
public:
  virtual void mf1()                   // forwarding
function; implicitly
  { Base::mf1(); }                     // inline (see Item
30) For info
  ...                                  // function, see item
34.)
};

...

Derived d;
int x;

d.mf1();                               // fine, calls
Derived::mf1
d.mf1(x);                              // error! Base::mf1()
is hidden
```

Another use for inline forwarding functions is to work around ancient compilers that (incorrectly) don't support `using` declarations to import inherited names into the scope of a derived class.

That's the whole story on inheritance and name hiding, but when inheritance is combined with templates, an entirely different form of the “inherited names are hidden” issue arises. For all the angle-bracket-demarcated details, see Item 43.

Things to Remember

- Names in derived classes hide names in base classes. Under public inheritance, this is never desirable.
- To make hidden names visible again, employ `using` declarations or forwarding functions.

Item 34: Differentiate between inheritance of interface and inheritance of implementation

The seemingly straightforward notion of (public) inheritance turns out, upon closer examination, to be composed of two separable parts: inheritance of function interfaces and inheritance of function implementations. The difference between these two kinds of inheritance corresponds exactly to the difference between function declarations and function definitions discussed in the Introduction to this book.

As a class designer, you sometimes want derived classes to inherit only the interface (declaration) of a member function. Sometimes you want derived classes to inherit both a function's interface and implementation, but you want to allow them to override the implementation they inherit. And sometimes you want derived classes to inherit a function's interface and implementation without allowing them to override anything.

To get a better feel for the differences among these options, consider a class hierarchy for representing geometric shapes in a graphics application:

Click here to view code image

```
class Shape {
public:
  virtual void draw() const = 0;

  virtual void error(const std::string& msg);

  int objectID() const;

  ...
};

class Rectangle: public Shape { ... };
class Ellipse: public Shape { ... };
```

`Shape` is an abstract class; its pure virtual function `draw` marks it as such. As a result, clients cannot create instances of the `Shape` class, only of classes derived from it. Nonetheless, `Shape` exerts a strong influence on all classes that (publicly) inherit from it, because

- Member function *interfaces are always inherited*. As explained in Item 32, public inheritance means is-a, so anything that is true of a base class must also be true of its derived classes. Hence, if a function applies to a class, it must also apply to its derived classes.

Three functions are declared in the `Shape` class. The first, `draw`, draws the current object on an implicit display. The second, `error`, is called when an error needs to be reported. The third, `objectID`, returns a unique integer identifier for the current object. Each function is declared in a different way: `draw` is a pure virtual function; `error` is a simple (impure?) virtual function; and `objectID` is a non-virtual function. What are the implications of these different declarations?

Consider first the pure virtual function `draw`:

```
class Shape {
public:
  virtual void draw() const = 0;
  ...
};
```

The two most salient features of pure virtual functions are that they *must* be redeclared by any concrete class that inherits them, and they typically have no definition in abstract classes. Put these two characteristics together, and you realize that

- The purpose of declaring a pure virtual function is to have derived classes inherit a function *interface only*.

This makes perfect sense for the `Shape::draw` function, because it is a reasonable demand that all `Shape` objects must be `draw`able, but the `Shape` class can provide no reasonable default implementation for that function. The algorithm for drawing an ellipse is very different from the algorithm for drawing a rectangle, for example. The declaration of `Shape::draw` says to designers of concrete derived classes, “You must provide a `draw` function, but

I have no idea how you're going to implement it."

Incidentally, it *is* possible to provide a definition for a pure virtual function. That is, you could provide an implementation for `Shape::draw`, and C++ wouldn't complain, but the only way to call it would be to qualify the call with the class name:

Click here to view code image

```
Shape *ps = new Shape;              // error! Shape is
abstract

Shape *ps1 = new Rectangle;         // fine
ps1->draw();                     // calls Rectangle::draw

Shape *ps2 = new Ellipse;           // fine
ps2->draw();                     // calls Ellipse::draw

ps1->Shape::draw();                 // calls Shape::draw

ps2->Shape::draw();                 // calls Shape::draw
```

Aside from helping you impress fellow programmers at cocktail parties, knowledge of this feature is generally of limited utility. As you'll see below, however, it can be employed as a mechanism for providing a safer-than-usual default implementation for simple (impure) virtual functions.

The story behind simple virtual functions is a bit different from that behind pure virtuals. As usual, derived classes inherit the interface of the function, but simple virtual functions provide an implementation that derived classes may override. If you think about this for a minute, you'll realize that

- The purpose of declaring a simple virtual function is to have derived classes inherit a function *interface as well as a default implementation.*

Consider the case of `Shape::error`:

Click here to view code image

```
class Shape {
public:
  virtual void error(const std::string& msg);
  ...
};
```

The interface says that every class must support a function to be called when an error is encountered, but each class is free to handle errors in whatever way it sees fit. If a class doesn't want to do anything special, it can just fall back on the default error handling provided in the `Shape` class. That is, the declaration of `Shape::error` says to designers of derived classes, “You've got to support an `error` function, but if you don't want to write your own, you can fall back on the default version in the `Shape` class.”

It turns out that it can be dangerous to allow simple virtual functions to specify both a function interface and a default implementation. To see why, consider a hierarchy of airplanes for XYZ Airlines. XYZ has only two kinds of planes, the Model A and the Model B, and both are flown in exactly the same way. Hence, XYZ designs the following hierarchy:

Click here to view code image

```
class Airport { ... };                     // represents
airports

class Airplane {
public:
  virtual void fly(const Airport& destination);

  ...

};

void Airplane::fly(const Airport& destination)
{
  default code for flying an airplane to the given
destination
}

class ModelA: public Airplane { ... };

class ModelB: public Airplane { ... };
```

To express that all planes have to support a `fly` function, and in recognition of the fact that different models of plane could, in principle, require different implementations for `fly`, `Airplane::fly` is declared virtual. However, in order to avoid writing identical code in the `ModelA` and `ModelB` classes, the default flying behavior is provided as the body of `Airplane::fly`, which both `ModelA` and `ModelB` inherit.

This is a classic object-oriented design. Two classes share a common feature (the way they implement `fly`), so the common feature is moved into a base class, and the feature is inherited by the two classes. This design makes common features explicit, avoids code duplication, facilitates future enhancements, and eases long-term maintenance — all the things for which object-oriented technology is so highly touted. XYZ Airlines should be proud.

Now suppose that XYZ, its fortunes on the rise, decides to acquire a new type of airplane, the Model C. The Model C differs in some ways from the Model A and the Model B. In particular, it is flown differently.

XYZ's programmers add the class for Model C to the hierarchy, but in their haste to get the new model into service, they forget to redefine the `fly` function:

Click here to view code image

```
class ModelC: public Airplane {

  ...                                    // no fly function is
declared
};
```

In their code, then, they have something akin to the following:

Click here to view code image

```
Airport PDX(...);                        // PDX is the airport
near my home

Airplane *pa = new ModelC;

...

pa->fly(PDX);                           // calls
Airplane::fly!
```

This is a disaster: an attempt is being made to fly a `ModelC` object as if it were a `ModelA` or a `ModelB`. That's not the kind of behavior that inspires confidence in the traveling public.

The problem here is not that `Airplane::fly` has default behavior, but that `ModelC` was allowed to inherit that behavior without explicitly saying that it wanted to. Fortunately, it's easy to offer default behavior to derived classes but not give it to them unless they ask for it. The trick is to sever the connection between the *interface* of the virtual function and its default *implementation*. Here's one way to do it:

Click here to view code image

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;

  ...

protected:
  void defaultFly(const Airport& destination);
};

void Airplane::defaultFly(const Airport& destination)
{
  default code for flying an airplane to the given
destination
}
```

Notice how `Airplane::fly` has been turned into a *pure* virtual function. That provides the interface for flying. The default implementation is also present in the `Airplane` class, but now it's in the form of an independent function, `defaultFly`. Classes like `ModelA` and `ModelB` that want to use the default behavior simply make an inline call to `defaultFly` inside their body of `fly` (but see Item 30 for information on the interaction of inlining and virtual functions):

Click here to view code image

```
class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { defaultFly(destination); }

  ...
};

class ModelB: public Airplane {
```

```
public:
  virtual void fly(const Airport& destination)
  { defaultFly(destination); }

  ...
};
```

For the `ModelC` class, there is no possibility of accidentally inheriting the incorrect implementation of `fly`, because the pure virtual in `Airplane` forces `ModelC` to provide its own version of `fly`.

Click here to view code image

```
class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);

  ...
};

void ModelC::fly(const Airport& destination)
{
  code for flying a ModelC airplane to the given destination
}
```

This scheme isn't foolproof (programmers can still copy-and-paste themselves into trouble), but it's more reliable than the original design. As for `Airplane::defaultFly`, it's protected because it's truly an implementation detail of `Airplane` and its derived classes. Clients using airplanes should care only that they can be flown, not how the flying is implemented.

It's also important that `Airplane::defaultFly` is a *non-virtual* function. This is because no derived class should redefine this function, a truth to which Item 36 is devoted. If `defaultFly` were virtual, you'd have a circular problem: what if some derived class forgets to redefine `defaultFly` when it's supposed to?

Some people object to the idea of having separate functions for providing interface and default implementation, such as `fly` and `defaultFly` above. For one thing, they note, it pollutes the class namespace with a proliferation of closely related function names. Yet they still agree that interface and default implementation should be separated. How do they resolve this seeming contradiction? By taking advantage of the fact that pure virtual functions must be redeclared in concrete derived classes, but they may also have implementations of their own. Here's how the `Airplane` hierarchy could take advantage of the ability to define a pure virtual function:

Click here to view code image

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;

  ...
};


void Airplane::fly(const Airport& destination)     // an
implementation of
{                                                  // a pure
virtual function
  default code for flying an airplane to
  the given destination
}

class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }

  ...

};

class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }

  ...

};

class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);

  ...
};

void ModelC::fly(const Airport& destination)
{
  code for flying a ModelC airplane to the given destination
}
```

This is almost exactly the same design as before, except that the body of the pure virtual function `Airplane::fly` takes the place of the independent function `Airplane::defaultFly`. In essence, `fly` has been broken into its two fundamental components. Its declaration specifies its interface (which derived classes *must* use), while its definition specifies its default behavior (which derived classes *may* use, but only if they explicitly request it). In merging `fly` and `defaultFly`, however, you've lost the ability to give the two functions different protection levels: the code that used to be protected (by being in `defaultFly`) is now public (because it's in `fly`).

Finally, we come to `Shape`'s non-virtual function, `objectID`:

```
class Shape {
public:
  int objectID() const;
  ...
};
```

When a member function is non-virtual, it's not supposed to behave differently in derived classes. In fact, a non-virtual member function specifies an *invariant over specialization*, because it identifies behavior that is not supposed to change, no matter how specialized a derived class becomes. As such,

- The purpose of declaring a non-virtual function is to have derived classes inherit a function *interface as well as a mandatory implementation*.

You can think of the declaration for `Shape::objectID` as saying, “Every `Shape` object has a function that yields an object identifier, and that object identifier is always computed the same way. That way is determined by the definition of `Shape::objectID`, and no derived class should try to change how it's done.” Because a non-virtual function identifies an *invariant* over specialization, it should never be redefined in a derived class, a point that is discussed in detail in Item 36.

The differences in declarations for pure virtual, simple virtual, and non-virtual functions allow you to specify with precision what you want derived classes to inherit: interface only, interface and a default implementation, or interface and a mandatory implementation, respectively. Because these different types of declarations mean fundamentally different things, you must choose carefully among them when you declare your member functions. If you do, you should avoid the two most common mistakes made by inexperienced class designers.

The first mistake is to declare all functions non-virtual. That leaves no room for specialization in derived classes; non-virtual destructors are particularly problematic (see Item 7). Of course, it's perfectly reasonable to design a class that is not intended to be used as a base class. In that case, a set of exclusively non-virtual member functions is appropriate. Too often, however, such classes are declared either out of ignorance of the differences between virtual and non-virtual functions or as a result of an unsubstantiated concern over the performance cost of virtual functions. The fact of the matter is that almost any class that's to be used as a base class will have virtual functions (again, see Item 7).

If you're concerned about the cost of virtual functions, allow me to bring up the empirically-based rule of 80-20 (see also Item 30), which states that in a typical program, 80% of the runtime will be spent executing just 20% of the code. This rule is important, because it means that, on average, 80% of your function calls can be virtual without having the slightest detectable impact on your program's overall performance. Before you go gray worrying about whether you can afford the cost of a virtual function, take the simple precaution of making sure that you're focusing on the 20% of your program where the decision might really make a difference.

The other common problem is to declare *all* member functions virtual. Sometimes this is the right thing to do — witness Item 31's Interface classes. However, it can also be a sign of a class designer who lacks the backbone to take a stand. Some functions should *not* be redefinable in derived classes, and whenever that's the case, you've got to say so by making those functions non-virtual. It serves no one to pretend that your class can be all things to all people if they'll just take the time to redefine all your functions. If you have an invariant over specialization, don't be afraid to say so!

Things to Remember

- Inheritance of interface is different from inheritance of implementation. Under public inheritance, derived classes always inherit base class interfaces.
- Pure virtual functions specify inheritance of interface only.
- Simple (impure) virtual functions specify inheritance of interface plus inheritance of a default implementation.
- Non-virtual functions specify inheritance of interface plus inheritance of a mandatory implementation.

Item 35: Consider alternatives to virtual functions

So you're working on a video game, and you're designing a hierarchy for characters in the game. Your game being of the slash-and-burn variety, it's not uncommon for characters to be injured or otherwise in a reduced state of health. You therefore decide to offer a member function, `healthValue`, that returns an integer indicating how healthy the character is. Because different characters may calculate their health in different ways, declaring `healthValue` virtual seems the obvious way to design things:

Click here to view code image

```
class GameCharacter {
public:
  virtual int healthValue() const;        // return
character's health rating;
  ...                                     // derived classes
may redefine this
};
```

The fact that `healthValue` isn't declared pure virtual suggests that there is a default algorithm for calculating health (see Item 34).

This is, indeed, the obvious way to design things, and in some sense, that's its weakness. Because this design is so obvious, you may not give adequate consideration to its alternatives. In the interest of helping you escape the ruts in the road of object-oriented design, let's consider some other ways to approach this problem.

The Template Method Pattern via the Non-Virtual Interface Idiom

We'll begin with an interesting school of thought that argues that virtual functions should almost always be private. Adherents to this school would suggest that a better design would retain `healthValue` as a public member function but make it non-virtual and have it call a private virtual function to do the real work, say, `doHealthValue`:

Click here to view code image

```
class GameCharacter {
public:
  int healthValue() const               // derived classes do
not redefine
  {                                     // this — see Item 36

    ...                                 // do "before" stuff
— see below

    int retVal = doHealthValue();       // do the real work

    ...                                 // do "after" stuff —
see below

    return retVal;
  }
  ...

private:
  virtual int doHealthValue() const     // derived classes
may redefine this
  {
    ...                                 // default algorithm
for calculating
  }                                     // character's health
};
```

In this code (and for the rest of this Item), I'm showing the bodies of member functions in class definitions. As Item 30 explains, that implicitly declares them `inline`. I'm showing the code this way only to make it easier to see what is going on. The designs I'm describing are independent of inlining decisions, so don't think it's meaningful that the member functions are defined inside classes. It's not.

This basic design — having clients call private virtual functions indirectly through public non-virtual member functions — is known as the *non-virtual interface (NVI) idiom*. It's a particular manifestation of the more general design pattern called Template Method (a pattern that, unfortunately, has nothing to do with C++ templates). I call the non-virtual function (e.g., `healthValue`) the virtual function's *wrapper*.

An advantage of the NVI idiom is suggested by the “do 'before' stuff” and “do 'after' stuff” comments in the code. Those comments identify code segments guaranteed to be called before and after the virtual function that does the real work. This means that the wrapper ensures that before a virtual function is called, the proper context is set up, and after the call is over, the context is cleaned up. For example, the “before” stuff could include locking a mutex, making a log entry, verifying that class invariants and function preconditions are satisfied, etc. The “after” stuff could include unlocking a mutex, verifying function postconditions, reverifying class invariants, etc. There's not really any good way to do that if you let clients call virtual functions directly.

It may have crossed your mind that the NVI idiom involves derived classes redefining private virtual functions — redefining functions they can't call! There's no design contradiction here. Redefining a virtual function specifies *how* something is to be done. Calling a virtual function specifies *when* it will be done. These concerns are independent. The NVI idiom allows derived classes to redefine a virtual function, thus giving them control over *how* functionality is implemented, but the base class reserves for itself the right to say *when* the function will be called. It may seem odd at first, but C++'s rule that derived classes may redefine private inherited virtual functions is perfectly sensible.

Under the NVI idiom, it's not strictly necessary that the virtual functions be private. In some class hierarchies, derived class implementations of a virtual function are expected to invoke their base class counterparts (e.g., the example on page 120), and for such calls to be legal, the virtuals must be protected, not private. Sometimes a virtual function even has to be public (e.g., destructors in polymorphic base classes — see Item 7), but then the NVI idiom can't really be applied.

The Strategy Pattern via Function Pointers

The NVI idiom is an interesting alternative to public virtual functions, but from a design point of view, it's little more than window dressing. After all, we're still using virtual functions to calculate each character's health. A more dramatic design assertion would be to say that calculating a character's health is independent of the character's type — that such calculations need not be part of the character at all. For example, we could require that each character's constructor be passed a pointer to a health calculation function, and we could call that function to do the actual calculation:

Click here to view code image

```
class GameCharacter;                              // forward
declaration

// function for the default health calculation algorithm
int defaultHealthCalc(const GameCharacter& gc);

class GameCharacter {
public:
  typedef int (*HealthCalcFunc)(const GameCharacter&);

  explicit GameCharacter(HealthCalcFunc hcf =
defaultHealthCalc)
  : healthFunc(hcf)
  {}

  int healthValue() const
  { return healthFunc(*this); }

  ...

private:
  HealthCalcFunc healthFunc;
};
```

This approach is a simple application of another common design pattern, Strategy. Compared to approaches based on virtual functions in the `GameCharacter` hierarchy, it offers some interesting flexibility:

- Different instances of the same character type can have different health calculation functions. For example:

Click here to view code image

```
class EvilBadGuy: public GameCharacter {
public:
  explicit EvilBadGuy(HealthCalcFunc hcf =
defaultHealthCalc)
  : GameCharacter(hcf)
  { ... }

  ...

};
int loseHealthQuickly(const GameCharacter&);     //
health calculation
int loseHealthSlowly(const GameCharacter&);      // funcs
with different
                                                 //
behavior

EvilBadGuy ebg1(loseHealthQuickly);              // same-
type charac-
EvilBadGuy ebg2(loseHealthSlowly);               // ters
with different
                                                 //
health-related
                                                 //
behavior
```

- Health calculation functions for a particular character may be changed at runtime. For example, `GameCharacter` might offer a member function, `setHealthCalculator`, that allowed replacement of the current health calculation function.

On the other hand, the fact that the health calculation function is no longer a member function of the `GameCharacter` hierarchy means that it has no special access to the internal parts of the object whose health it's calculating. For example, `defaultHealthCalc` has no access to the non-public parts of `EvilBadGuy`. If a character's health can be calculated based purely on information available through the character's public interface, this is not a problem, but if accurate health calculation requires non-public information, it is. In fact, it's a potential issue anytime you replace functionality inside a class (e.g., via a member function) with equivalent functionality outside the class (e.g., via a non-member non-friend function or via a non-friend member function of another class). This issue will persist for the remainder of this Item, because all the other design alternatives we're going to consider involve the use of functions outside the `GameCharacter` hierarchy.

As a general rule, the only way to resolve the need for non-member functions to have access to non-public parts of a class is to weaken the class's encapsulation. For example, the class might declare the non-member functions to be `friend`s, or it might offer public accessor functions for parts of its implementation it would otherwise prefer to keep hidden. Whether the advantages of using a function pointer instead of a virtual function (e.g., the ability to have per-object health calculation functions and the ability to change such functions at runtime) offset the possible need to decrease `GameCharacter`'s encapsulation is something you must decide on a design-by-design basis.

The Strategy Pattern via `tr1::function`

Once you accustom yourself to templates and their use of implicit interfaces (see Item 41), the function-pointer-based approach looks rather rigid. Why must the health calculator be a function instead of simply something that *acts* like a function (e.g., a function object)? If it must be a function, why can't it be a member function? And why must it return an `int` instead of any type *convertible* to an `int`?

These constraints evaporate if we replace the use of a function pointer (such as `healthFunc`) with an object of type `tr1::function`. As Item 54 explains, such objects may hold *any callable entity* (i.e., function pointer, function object, or member function pointer) whose signature is compatible with what is expected. Here's the design we just saw, this time using `tr1::function`:

Click here to view code image

```
class GameCharacter;                                 // as
before
int defaultHealthCalc(const GameCharacter& gc);      // as
before

class GameCharacter {
public:
   // HealthCalcFunc is any callable entity that can be
called with
   // anything compatible with a GameCharacter and that
returns anything
   // compatible with an int; see below for details
   typedef std::tr1::function<int (const GameCharacter&)>
HealthCalcFunc;

   explicit GameCharacter(HealthCalcFunc hcf =
defaultHealthCalc)
   : healthFunc(hcf)
   {}

   int healthValue() const
   { return healthFunc(*this);   }

   ...

private:
  HealthCalcFunc healthFunc;
};
```

As you can see, `HealthCalcFunc` is a typedef for a `tr1::function` instantiation. That means it acts like a generalized function pointer type. Look closely at what `HealthCalcFunc` is a typedef for:

```
std::tr1::function<int (const GameCharacter&)>
```

Here I've highlighted the “target signature” of this `tr1::function` instantiation. That target signature is “function taking a `const GameCharacter&` and returning an `int`.” An object of this `tr1::function` type (i.e., of type `HealthCalcFunc`) may hold any callable entity compatible with the target signature. To be compatible means that `const GameCharacter&` either is or can be converted to the type of the entity’s parameter, and the entity’s return type either is or can be implicitly converted to `int`.

Compared to the last design we saw (where `GameCharacter` held a pointer to a function), this design is almost the same. The only difference is that `GameCharacter` now holds a `tr1::function` object — a *generalized* pointer to a function. This change is so small, I'd call it inconsequential, except that a consequence is that clients now have staggeringly more flexibility in specifying health calculation functions:

Click here to view code image

```
short calcHealth(const GameCharacter&);          // health
calculation
                                                 // function;
```

```
note
                                                 // non-int
return type

struct HealthCalculator {                        // class for
health
int operator()(const GameCharacter&) const       //
calculation function
  { ... }                                        // objects
};

class GameLevel {
public:
  float health(const GameCharacter&) const;      // health
calculation
  ...                                            // mem
function; note
};                                               // non-int
return type


class EvilBadGuy: public GameCharacter {         // as before
  ...
};

class EyeCandyCharacter:   public GameCharacter {  // another
character
  ...                                              // type;
assume same
};                                                 //
constructor as
                                                   //
EvilBadGuy


EvilBadGuy ebg1(calcHealth);                       //
character using a
                                                   // health
calculation
                                                   //
function


EyeCandyCharacter ecc1(HealthCalculator());        //
character using a
```

```
                                                   // health
calculation
                                                   //
function object

GameLevel currentLevel;
...
EvilBadGuy ebg2(                                   //
character using a
  std::tr1::bind(&GameLevel::health,               // health
calculation
          currentLevel,                            // member
function;
          _1)                                      // see
below for details
);
```

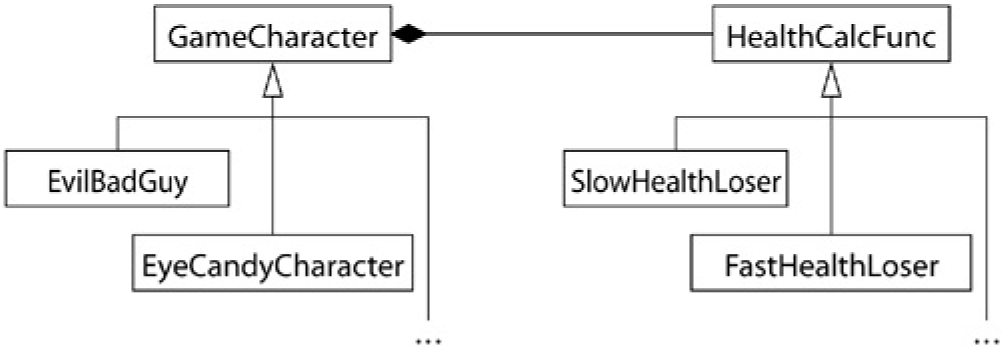
Personally, I find what `tr1::function` lets you do so amazing, it makes me tingle all over. If you're not tingling, it may be because you're staring at the definition of `ebg2` and wondering what's going on with the call to `tr1::bind`. Kindly allow me to explain.

We want to say that to calculate `ebg2`'s health rating, the `health` member function in the `GameLevel` class should be used. Now, `GameLevel::health` is a function that is declared to take one parameter (a reference to a `GameCharacter`), but it really takes two, because it also gets an implicit `GameLevel` parameter — the one `this` points to. Health calculation functions for `GameCharacter`s, however, take a single parameter: the `GameCharacter` whose health is to be calculated. If we're to use `GameLevel::health` for `ebg2`'s health calculation, we have to somehow "adapt" it so that instead of taking two parameters (a `GameCharacter` and a `GameLevel`), it takes only one (a `GameCharacter`). In this example, we always want to use `currentLevel` as the `GameLevel` object for `ebg2`'s health calculation, so we "bind" `currentLevel` as the `GameLevel` object to be used each time `GameLevel::health` is called to calculate `ebg2`'s health. That's what the `tr1::bind` call does: it specifies that `ebg2`'s health calculation function should always use `currentLevel` as the `GameLevel` object.

I'm skipping over a host of details regarding the call to `tr1::bind`, because such details wouldn't be terribly illuminating, and they'd distract from the fundamental point I want to make: by using `tr1::function` instead of a function pointer, we're allowing clients to use *any compatible callable entity* when calculating a character's health. Is that cool or what?

The “Classic” Strategy Pattern

If you're more into design patterns than C++ coolness, a more conventional approach to Strategy would be to make the health-calculation function a virtual member function of a separate health-calculation hierarchy. The resulting hierarchy design would look like this:



If you're not up on your UML notation, this just says that `GameCharacter` is the root of an inheritance hierarchy where `EvilBadGuy` and `EyeCandyCharacter` are derived classes; `HealthCalcFunc` is the root of an inheritance hierarchy with derived classes `SlowHealthLoser` and `FastHealthLoser`; and each object of type `GameCharacter` contains a pointer to an object from the `HealthCalcFunc` hierarchy.

Here's the corresponding code skeleton:

Click here to view code image

```
class GameCharacter;                               // forward
declaration

class HealthCalcFunc {
public:

  ...
  virtual int calc(const GameCharacter& gc) const
  { ... }
  ...

};

HealthCalcFunc defaultHealthCalc;

class GameCharacter {
public:
  explicit GameCharacter(HealthCalcFunc *phcf =
&defaultHealthCalc)
  : pHealthCalc(phcf)
  {}

  int healthValue() const
  { return pHealthCalc->calc(*this);}

  ...

private:
  HealthCalcFunc *pHealthCalc;
};
```

This approach has the appeal of being quickly recognizable to people familiar with the “standard” Strategy pattern implementation, plus it offers the possibility that an existing health calculation algorithm can be tweaked by adding a derived class to the `HealthCalcFunc` hierarchy.

Summary

The fundamental advice of this Item is to consider alternatives to virtual functions when searching for a design for the problem you're trying to solve. Here's a quick recap of the alternatives we examined:

- Use the **non-virtual interface idiom** (NVI idiom), a form of the Template Method design pattern that wraps public non-virtual member functions around less accessible virtual functions.
- Replace virtual functions with **function pointer data members**, a stripped-down manifestation of the Strategy design pattern.
- Replace virtual functions with **tr1::function data members**, thus allowing use of any callable entity with a signature compatible with what you need. This, too, is a form of the Strategy design pattern.
- Replace virtual functions in one hierarchy with **virtual functions in another hierarchy**. This is the conventional implementation of the Strategy design pattern.

This isn't an exhaustive list of design alternatives to virtual functions, but it should be enough to convince you that there *are* alternatives. Furthermore, their comparative advantages and disadvantages should make clear that you *should* consider them.

To avoid getting stuck in the ruts of the road of object-oriented design, give the wheel a good jerk from time to time. There are lots of other roads. It's worth taking the time to investigate them.

Things to Remember

- Alternatives to virtual functions include the NVI idiom and various forms of the Strategy design pattern. The NVI idiom is itself an example of the Template Method design pattern.
- A disadvantage of moving functionality from a member function to a function outside the class is that the non-member function lacks access to the class's non-public members.
- `tr1::function` objects act like generalized function pointers. Such objects support all callable entities compatible with a given target signature.

Item 36: Never redefine an inherited non-virtual function

Suppose I tell you that a class `D` is publicly derived from a class `B` and that there is a public member function `mf` defined in class `B`. The parameters and return type of `mf` are unimportant, so let's just assume they're both `void`. In other words, I say this:

```
class B {
public:
  void mf();
  ...
};
class D: public B { ... };
```

Even without knowing anything about `B`, `D`, or `mf`, given an object `x` of type `D`,

```
D x;                              // x is an object of type D
```

you would probably be quite surprised if this,

Click here to view code image

```
B *pB = &x;                       // get pointer to x
```

```
pB->mf();                          // call mf through pointer
```

behaved differently from this:

Click here to view code image

```
D *pD = &x;                        // get pointer to x
pD->mf();                          // call mf through pointer
```

That's because in both cases you're invoking the member function `mf` on the object `x`. Because it's the same function and the same object in both cases, it should behave the same way, right?

Right, it should. But it might not. In particular, it won't if `mf` is non-virtual and `D` has defined its own version of `mf`:

Click here to view code image

```
class D: public B {
public:
  void mf();                       // hides B::mf; see Item 33

  ...

};

pB->mf();                          // calls B::mf

pD->mf();                          // calls D::mf
```

The reason for this two-faced behavior is that *non-virtual* functions like `B::mf` and `D::mf` are statically bound (see Item 37). That means that because `pB` is declared to be of type pointer-to-`B`, non-virtual functions invoked through `pB` will *always* be those defined for class `B`, even if `pB` points to an object of a class derived from `B`, as it does in this example.

Virtual functions, on the other hand, are dynamically bound (again, see Item 37), so they don't suffer from this problem. If `mf` were a virtual function, a call to `mf` through either `pB` or `pD` would result in an invocation of `D::mf`, because what `pB` and `pD` *really* point to is an object of type `D`.

If you are writing class `D` and you redefine a non-virtual function `mf` that you inherit from class `B`, `D` objects will likely exhibit inconsistent behavior. In particular, any given `D` object may act like either a `B` or a `D` when `mf` is called, and the determining factor will have nothing to do with the object itself, but with the declared type of the pointer that points to it. References exhibit the same baffling behavior as do pointers.

But that's just a pragmatic argument. What you really want, I know, is some kind of theoretical justification for not redefining inherited non-virtual functions. I am pleased to oblige.

Item 32 explains that public inheritance means is-a, and Item 34 describes why declaring a non-virtual function in a class establishes an invariant over specialization for that class. If you apply these observations to the classes `B` and `D` and to the non-virtual member function `B::mf`, then

- Everything that applies to `B` objects also applies to `D` objects, because every `D` object is-a `B` object;
- Classes derived from `B` must inherit both the interface *and* the implementation of `mf`, because `mf` is non-virtual in `B`.

Now, if `D` redefines `mf`, there is a contradiction in your design. If `D` *really* needs to implement `mf` differently from `B`, and if every `B` object — no matter how specialized — *really* has to use the `B` implementation for `mf`, then it's simply not true that every `D` is-a `B`. In that case, `D` shouldn't publicly inherit from `B`. On the other hand, if `D` *really* has to publicly inherit from `B`, and if `D` *really* needs to implement `mf` differently from `B`, then it's just not true that `mf` reflects an invariant over specialization for `B`. In that case, `mf` should be virtual. Finally, if every `D` *really* is-a `B`, and if `mf` really corresponds to an invariant over specialization for `B`, then `D` can't honestly need to redefine `mf`, and it shouldn't try to.

Regardless of which argument applies, something has to give, and under no conditions is it the prohibition on redefining an inherited non-virtual function.

If reading this Item gives you a sense of *déjà vu*, it's probably because you've already read Item 7, which explains why destructors in polymorphic base classes should be virtual. If you violate that guideline (i.e., if you declare a non-virtual destructor in a polymorphic base class), you'll also be violating this guideline, because derived classes would invariably redefine an inherited non-virtual function: the base class's destructor. This would be true even for derived classes that declare no destructor, because, as Item 5 explains, the destructor is one of the member functions that compilers generate for you if you don't declare one yourself. In essence, Item 7 is nothing more than a special case of this Item, though it's important enough to merit calling out on its own.

Things to Remember

- Never redefine an inherited non-virtual function.

Item 37: Never redefine a function's inherited default parameter value

Let's simplify this discussion right from the start. There are only two kinds of functions you can inherit: virtual and non-virtual. However, it's always a mistake to redefine an inherited non-virtual function (see Item 36), so we can safely limit our discussion here to the situation in which you inherit a *virtual* function with a default parameter value.

That being the case, the justification for this Item becomes quite straightforward: virtual functions are dynamically bound, but default parameter values are statically bound.

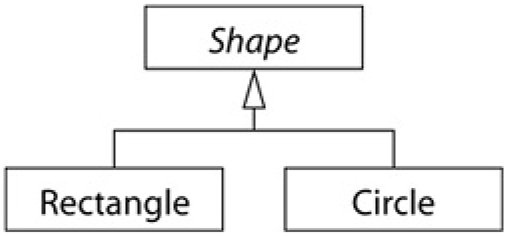
What's that? You say the difference between static and dynamic binding has slipped your already overburdened mind? (For the record, static binding is also known as *early binding*, and dynamic binding is also known as *late binding*.) Let's review, then.

An object's *static type* is the type you declare it to have in the program text. Consider this class hierarchy:

Click here to view code image

```
// a class for geometric shapes
class Shape {
public:
  enum ShapeColor { Red, Green, Blue };

  // all shapes must offer a function to draw themselves
  virtual void draw(ShapeColor color = Red) const = 0;
  ...
};

class Rectangle: public Shape {
public:
  // notice the different default parameter value — bad!
  virtual void draw(ShapeColor color = Green) const;
  ...
};

class Circle: public Shape {
public:
  virtual void draw(ShapeColor color) const;
  ...
};
```

Graphically, it looks like this:



Now consider these pointers:

Click here to view code image

```
Shape *ps;                        // static type = Shape*
Shape *pc = new Circle;           // static type = Shape*
Shape *pr = new Rectangle;        // static type = Shape*
```

In this example, `ps`, `pc`, and `pr` are all declared to be of type pointer-to-`Shape`, so they all have that as their static type. Notice that it makes absolutely no difference what they're *really* pointing to — their static type is `Shape*` regardless.

An object's *dynamic type* is determined by the type of the object to which it currently refers. That is, its dynamic type indicates how it will behave. In the example above, `pc`'s dynamic type is `Circle*`, and `pr`'s dynamic type is `Rectangle*`. As for `ps`, it doesn't really have a dynamic type, because it doesn't refer to any object (yet).

Dynamic types, as their name suggests, can change as a program runs, typically through assignments:

Click here to view code image

```
ps = pc;                        // ps's dynamic type is
                                // now Circle*
```

```
ps = pr;                         // ps's dynamic type is
                                 // now Rectangle*
```

Virtual functions are *dynamically bound*, meaning that the particular function called is determined by the dynamic type of the object through which it's invoked:

[Click here to view code image](#)

```
pc->draw(Shape::Red);               // calls
Circle::draw(Shape::Red)

pr->draw(Shape::Red);               // calls
Rectangle::draw(Shape::Red)
```

This is all old hat, I know; you surely understand virtual functions. The twist comes in when you consider virtual functions with default parameter values, because, as I said above, virtual functions are dynamically bound, but default parameters are statically bound. That means you may end up invoking a virtual function defined in a *derived class* but using a default parameter value from a *base class*:

```
pr->draw();                         // calls
Rectangle::draw(Shape::Red)!
```

In this case, `pr`'s dynamic type is `Rectangle*`, so the `Rectangle` virtual function is called, just as you would expect. In `Rectangle::draw`, the default parameter value is `Green`. Because `pr`'s static type is `Shape*`, however, the default parameter value for this function call is taken from the `Shape` class, not the `Rectangle` class! The result is a call consisting of a strange and almost certainly unanticipated combination of the declarations for `draw` in both the `Shape` and `Rectangle` classes.

The fact that `ps`, `pc`, and `pr` are pointers is of no consequence in this matter. Were they references, the problem would persist. The only important things are that `draw` is a virtual function, and one of its default parameter values is redefined in a derived class.

Why does C++ insist on acting in this perverse manner? The answer has to do with runtime efficiency. If default parameter values were dynamically bound, compilers would have to come up with a way to determine the appropriate default value(s) for parameters of virtual functions at runtime, which would be slower and more complicated than the current mechanism of determining them during compilation. The decision was made to err on the side of speed and simplicity of implementation, and the result is that you now enjoy execution behavior that is efficient, but, if you fail to heed the advice of this Item, confusing.

That's all well and good, but look what happens if you try to follow this rule and also offer default parameter values to users of both base and derived classes:

Click here to view code image

```
class Shape {
public:
  enum ShapeColor { Red, Green, Blue };

  virtual void draw(ShapeColor color = Red) const = 0;
  ...
};

class Rectangle: public Shape {
public:
  virtual void draw(ShapeColor color = Red) const;
  ...
};
```

Uh oh, code duplication. Worse yet, code duplication with dependencies: if the default parameter value is changed in `Shape`, all derived classes that repeat it must also be changed. Otherwise they'll end up redefining an inherited default parameter value. What to do?

When you're having trouble making a virtual function behave the way you'd like, it's wise to consider alternative designs, and Item 35 is filled with alternatives to virtual functions. One of the alternatives is the *non-virtual interface idiom* (NVI idiom): having a public non-virtual function in a base class call a private virtual function that derived classes may redefine. Here, we have the non-virtual function specify the default parameter, while the virtual function does the actual work:

Click here to view code image

```
class Shape {
public:
  enum ShapeColor { Red, Green, Blue };

  void draw(ShapeColor color = Red) const           // now
non-virtual
  {
    doDraw(color);                                  // calls
a virtual
  }

  ...

private:
  virtual void doDraw(ShapeColor color) const = 0;  // the
actual work is
};                                                  // done
in this func

class Rectangle: public Shape {
public:

  ...

private:
  virtual void doDraw(ShapeColor color) const;       // note
lack of a
  ...                                                //
default param val.
};
```

Because non-virtual functions should ovnever be overridden by derived classes (see Item 36), this design makes clear that the default value for `draw`'s `color` parameter should always be `Red`.

Things to Remember

- Never redefine an inherited default parameter value, because default parameter values are statically bound, while virtual functions — the only functions you should be overriding — are dynamically bound.

Item 38: Model “has-a” or “is-implemented-in-terms-

of” through composition

Composition is the relationship between types that arises when objects of one type contain objects of another type. For example:

Click here to view code image

```
class Address { ... };             // where someone lives

class PhoneNumber { ... };

class Person {
public:
  ...

private:
  std::string name;               // composed object
  Address address;                // ditto
  PhoneNumber voiceNumber;        // ditto
  PhoneNumber faxNumber;          // ditto
};
```

In this example, `Person` objects are composed of `string`, `Address`, and `PhoneNumber` objects. Among programmers, the term *composition* has lots of synonyms. It's also known as *layering*, *containment*, *aggregation*, and *embedding*.

Item 32 explains that public inheritance means “is-a.” Composition has a meaning, too. Actually, it has two meanings. Composition means either “has-a” or “is-implemented-in-terms-of.” That's because you are dealing with two different domains in your software. Some objects in your programs correspond to things in the world you are modeling, e.g., people, vehicles, video frames, etc. Such objects are part of the *application domain*. Other objects are purely implementation artifacts, e.g., buffers, mutexes, search trees, etc. These kinds of objects correspond to your software's *implementation domain*. When composition occurs between objects in the application domain, it expresses a has-a relationship. When it occurs in the implementation domain, it expresses an is-implemented-in-terms-of relationship.

The `Person` class above demonstrates the has-a relationship. A `Person` object has a name, an address, and voice and fax telephone numbers. You wouldn't say that a person *is* a name or that a person *is* an address. You would say that a person *has* a name and *has* an address. Most people have little difficulty with this distinction, so confusion between the roles of is-a and has-a is relatively rare.

Somewhat more troublesome is the difference between is-a and is-implemented-in-terms-of. For example, suppose you need a template for classes representing fairly small sets of objects, i.e., collections without duplicates. Because reuse is a wonderful thing, your first instinct is to employ the standard library's `set` template. Why write a new template when you can use one that's already been written?

Unfortunately, `set` implementations typically incur an overhead of three pointers per element. This is because `set`s are usually implemented as balanced search trees, something that allows them to guarantee logarithmic-time lookups, insertions, and erasures. When speed is more important than space, this is a reasonable design, but it turns out that for your application, space is more important than speed. The standard library's `set` thus offers the wrong trade-off for you. It seems you'll need to write your own template after all.

Still, reuse *is* a wonderful thing. Being the data structure maven you are, you know that of the many choices for implementing sets, one is to use linked lists. You also know that the standard C++ library has a `list` template, so you decide to (re)use it.

In particular, you decide to have your nascent `Set` template inherit from `list`. That is, `Set<T>` will inherit from `list<T>`. After all, in your implementation, a `Set` object will in fact *be* a `list` object. You thus declare your `Set` template like this:

```
template<typename T>                       // the wrong way
to use list for Set
class Set: public std::list<T> { ... };
```

Everything may seem fine at this point, but in fact there is something quite wrong. As Item 32 explains, if `D` is-a `B`, everything true of `B` is also true of `D`. However, a `list` object may contain duplicates, so if the value 3051 is inserted into a `list<int>` twice, that list will contain two copies of 3051. In contrast, a `Set` may not contain duplicates, so if the value 3051 is inserted into a `Set<int>` twice, the set contains only one copy of the value. It is thus untrue that a `Set` is-a `list`, because some of the things that are true for `list` objects are not true for `Set` objects.

Because the relationship between these two classes isn't is-a, public inheritance is the wrong way to model that relationship. The right way is to realize that a `Set` object can be *implemented in terms of* a `list` object:

Click here to view code image

```
template<class T>                    // the right way to use
list for Set
class Set {
public:
  bool member(const T& item) const;

  void insert(const T& item);
  void remove(const T& item);

  std::size_t size() const;

private:
  std::list<T> rep;                  // representation for Set
data
};
```

`Set`'s member functions can lean heavily on functionality already offered by `list` and other parts of the standard library, so the implementation is straightforward, as long as you're familiar with the basics of programming with the STL:

Click here to view code image

```
template<typename T>
bool Set<T>::member(const T& item) const
{
  return std::find(rep.begin(), rep.end(), item) !=
rep.end();
}
template<typename T>
void Set<T>::insert(const T& item)
{
  if (!member(item)) rep.push_back(item);
}
template<typename T>
```

```
void Set<T>::remove(const T& item)
{
  typename std::list<T>::iterator it =               // see
Item 42 for info on
    std::find(rep.begin(), rep.end(), item);         //
"typename" here
  if (it != rep.end()) rep.erase(it);
}
template<typename T>
std::size_t Set<T>::size() const
{
  return rep.size();
}
```

These functions are simple enough that they make reasonable candidates for inlining, though I know you'd want to review the discussion in Item 30 before making any firm inlining decisions.

One can argue that `Set`'s interface would be more in accord with Item 18's admonition to design interfaces that are easy to use correctly and hard to use incorrectly if it followed the STL container conventions, but following those conventions here would require adding a lot of stuff to `Set` that would obscure the relationship between it and `list`. Since that relationship is the point of this Item, we'll trade STL compatibility for pedagogical clarity. Besides, nits about `Set`'s interface shouldn't overshadow what's indisputably right about `Set`: the relationship between it and `list`. That relationship is not is-a (though it initially looked like it might be), it's is-implemented-in-terms-of.

Things to Remember

- Composition has meanings completely different from that of public inheritance.
- In the application domain, composition means has-a. In the implementation domain, it means is-implemented-in-terms-of.

Item 39: Use private inheritance judiciously

Item 32 demonstrates that C++ treats public inheritance as an is-a relationship. It does this by showing that compilers, when given a hierarchy in which a class `Student` publicly inherits from a class `Person`, implicitly convert `Student`s to `Person`s when that is necessary for a function call to succeed. It's worth repeating a portion of that example using private inheritance instead of public inheritance:

Click here to view code image

```
class Person { ... };
class Student: private Person { ... };     // inheritance is
now private

void eat(const Person& p);                 // anyone can eat

void study(const Student& s);              // only students
study

Person p;                                  // p is a Person
Student s;                                 // s is a Student

eat(p);                                    // fine, p is a
Person

eat(s);                                    // error! a
Student isn't a Person
```

Clearly, private inheritance doesn't mean is-a. What does it mean then?

“Whoa!” you say. “Before we get to the meaning, let's cover the behavior. How does private inheritance behave?” Well, the first rule governing private inheritance you've just seen in action: in contrast to public inheritance, compilers will generally *not* convert a derived class object (such as `Student`) into a base class object (such as `Person`) if the inheritance relationship between the classes is private. That's why the call to `eat` fails for the object `s`. The second rule is that members inherited from a private base class become private members of the derived class, even if they were protected or public in the base class.

So much for behavior. That brings us to meaning. Private inheritance means is-implemented-in-terms-of. If you make a class `D` privately inherit from a class `B`, you do so because you are interested in taking advantage of some of the features available in class `B`, not because there is any conceptual relationship between objects of types `B` and `D`. As such, private inheritance is purely an implementation technique. (That's why everything you inherit from a private base class becomes private in your class: it's all just implementation detail.) Using the terms introduced in Item 34, private inheritance means that implementation *only* should be inherited; interface should be ignored. If `D` privately inherits from `B`, it means that `D` objects are implemented in terms of `B` objects, nothing more. Private inheritance means nothing during software *design*, only during software *implementation*.

The fact that private inheritance means is-implemented-in-terms-of is a little disturbing, because Item 38 points out that composition can mean the same thing. How are you supposed to choose between them? The answer is simple: use composition whenever you can, and use private inheritance whenever you must. When must you? Primarily when protected members and/or virtual functions enter the picture, though there's also an edge case where space concerns can tip the scales toward private inheritance. We'll worry about the edge case later. After all, it's an edge case.

Suppose we're working on an application involving `Widget`s, and we decide we need to better understand how `Widget`s are being used. For example, not only do we want to know things like how often `Widget` member functions are called, we also want to know how the call ratios change over time. Programs with distinct phases of execution can have different behavioral profiles during the different phases. For example, the functions used during the parsing phase of a compiler are largely different from the functions used during optimization and code generation.

We decide to modify the `Widget` class to keep track of how many times each member function is called. At runtime, we'll periodically examine that information, possibly along with the values of each `Widget` and whatever other data we deem useful. To make this work, we'll need to set up a timer of some kind so that we'll know when it's time to collect the usage statistics.

Preferring to reuse existing code over writing new code, we rummage around in our utility toolkit and are pleased to find the following class:

Click here to view code image

```
class Timer {
public:
  explicit Timer(int tickFrequency);
   virtual void onTick() const;          // automatically
called for each tick
  ...
};
```

This is just what we're looking for. A `Timer` object can be configured to tick with whatever frequency we need, and on each tick, it calls a virtual function. We can redefine that virtual function so that it examines the current state of the `Widget` world. Perfect!

In order for `Widget` to redefine a virtual function in `Timer`, `Widget` must inherit from `Timer`. But public inheritance is inappropriate in this case. It's not true that a `Widget` is-a `Timer`. `Widget` clients shouldn't be able to call `onTick` on a `Widget`, because that's not part of the conceptual `Widget` interface. Allowing such a function call would make it easy for clients to use the `Widget` interface incorrectly, a clear violation of Item 18's advice to make interfaces easy to use correctly and hard to use incorrectly. Public inheritance is not a valid option here.

We thus inherit privately:

Click here to view code image

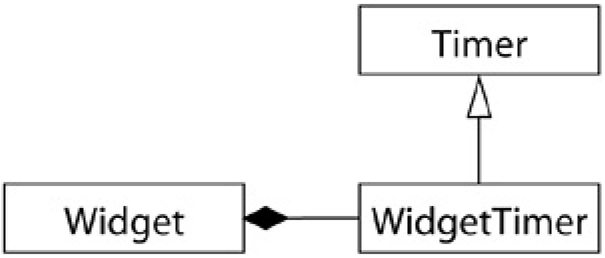
```
class Widget: private Timer {
private:
  virtual void onTick() const;           // look at Widget
usage data, etc.
  ...
};
```

By virtue of private inheritance, `Timer`'s public `onTick` function becomes private in `Widget`, and we keep it there when we redeclare it. Again, putting `onTick` in the public interface would mislead clients into thinking they could call it, and that would violate Item 18.

This is a nice design, but it's worth noting that private inheritance isn't strictly necessary. If we were determined to use composition instead, we could. We'd just declare a private nested class inside `Widget` that would publicly inherit from `Timer`, redefine `onTick` there, and put an object of that type inside `Widget`. Here's a sketch of the approach:

Click here to view code image

```
class Widget {
private:
  class WidgetTimer: public Timer {
  public:
    virtual void onTick() const;
    ...
  };
   WidgetTimer timer;
  ...
};
```



This design is more complicated than the one using only private inheritance, because it involves both (public) inheritance and composition, as well as the introduction of a new class (`WidgetTimer`). To be honest, I show it primarily to remind you that there is more than one way to approach a design problem, and it's worth training yourself to consider multiple approaches (see also Item 35). Nevertheless, I can think of two reasons why you might prefer public inheritance plus composition over private inheritance.

First, you might want to design `Widget` to allow for derived classes, but you might also want to prevent derived classes from redefining `onTick`. If `Widget` inherits from `Timer`, that's not possible, not even if the inheritance is private. (Recall from Item 35 that derived classes may redefine virtual functions even if they are not permitted to call them.) But if `WidgetTimer` is private in `Widget` and inherits from `Timer`, `Widget`'s derived classes have no access to `WidgetTimer`, hence can't inherit from it or redefine its virtual functions. If you've programmed in Java or C# and miss the ability to prevent derived classes from redefining virtual functions (i.e., Java's `final` methods and C#'s `sealed` ones), now you have an idea how to approximate that behavior in C++.

Second, you might want to minimize `Widget`'s compilation dependencies. If `Widget` inherits from `Timer`, `Timer`'s definition must be available when `Widget` is compiled, so the file defining `Widget` probably has to `#include Timer.h`. On the other hand, if `WidgetTimer` is moved out of `Widget` and `Widget` contains only a pointer to a `WidgetTimer`, `Widget` can get by with a simple declaration for the `WidgetTimer` class; it need not `#include` anything to do with `Timer`. For large systems, such decouplings can be important. (For details on minimizing compilation dependencies, consult Item 31.)

I remarked earlier that private inheritance is useful primarily when a would-be derived class wants access to the protected parts of a would-be base class or would like to redefine one or more of its virtual functions, but the conceptual relationship between the classes is is-implemented-in-terms-of instead of is-a. However, I also said that there was an edge case involving space optimization that could nudge you to prefer private inheritance over composition.

The edge case is edgy indeed: it applies only when you're dealing with a class that has no data in it. Such classes have no non-static data members; no virtual functions (because the existence of such functions adds a `vptr` to each object — see Item 7); and no virtual base classes (because such base classes also incur a size overhead — see Item 40). Conceptually, objects of such *empty classes* should use no space, because there is no per-object data to be stored. However, there are technical reasons for C++ decreeing that freestanding objects must have non-zero size, so if you do this,

Click here to view code image

```
class Empty {};                     // has no data, so
objects should
                                    // use no memory
class HoldsAnInt {                  // should need only
space for an int
private:
  int x;
  Empty e;                          // should require no
memory
};
```

you'll find that `sizeof(HoldsAnInt) > sizeof(int)`; an `Empty` data member requires memory. With most compilers, `sizeof(Empty)` is 1, because C++'s edict against zero-size freestanding objects is typically satisfied by the silent insertion of a `char` into “empty” objects. However, alignment requirements (see Item 50) may cause compilers to add padding to classes like `HoldsAnInt`, so it's likely that `HoldsAnInt` objects wouldn't gain just the size of a `char`, they would actually enlarge enough to hold a second `int`. (On all the compilers I tested, that's exactly what happened.)

But perhaps you've noticed that I've been careful to say that “freestanding” objects mustn't have zero size. This constraint doesn't apply to base class parts of derived class objects, because they're not freestanding. If you *inherit* from `Empty` instead of containing an object of that type,

```
class HoldsAnInt: private Empty {
private:
  int x;
};
```

you're almost sure to find that `sizeof(HoldsAnInt) == sizeof(int)`. This is known as the *empty base optimization* (EBO), and it's implemented by all the compilers I tested. If you're a library developer whose clients care about space, the EBO is worth knowing about. Also worth knowing is that the EBO is generally viable only under single inheritance. The rules governing C++ object layout generally mean that the EBO can't be applied to derived classes that have more than one base.

In practice, “empty” classes aren't truly empty. Though they never have non-static data members, they often contain typedefs, enums, static data members, or non-virtual functions. The STL has many technically empty classes that contain useful members (usually typedefs), including the base classes `unary_function` and `binary_function`, from which classes for user-defined function objects typically inherit. Thanks to widespread implementation of the EBO, such inheritance rarely increases the size of the inheriting classes.

Still, let's get back to basics. Most classes aren't empty, so the EBO is rarely a legitimate justification for private inheritance. Furthermore, most inheritance corresponds to is-a, and that's a job for public inheritance, not private. Both composition and private inheritance mean is-implemented-in-terms-of, but composition is easier to understand, so you should use it whenever you can.

Private inheritance is most likely to be a legitimate design strategy when you're dealing with two classes not related by is-a where one either needs access to the protected members of another or needs to redefine one or more of its virtual functions. Even in that case, we've seen that a mixture of public inheritance and containment can often yield the behavior you want, albeit with greater design complexity. Using private inheritance *judiciously* means employing it when, having considered all the alternatives, it's the best way to express the relationship between two classes in your software.

Things to Remember

- Private inheritance means is-implemented-in-terms of. It's usually inferior to composition, but it makes sense when a derived class needs access to protected base class members or needs to redefine inherited virtual functions.

- Unlike composition, private inheritance can enable the empty base optimization. This can be important for library developers who strive to minimize object sizes.

Item 40: Use multiple inheritance judiciously

When it comes to multiple inheritance (MI), the C++ community largely breaks into two basic camps. One camp believes that if single inheritance (SI) is good, multiple inheritance must be better. The other camp argues that single inheritance is good, but multiple inheritance isn't worth the trouble. In this Item, our primary goal is to understand both perspectives on the MI question.

One of the first things to recognize is that when MI enters the designscape, it becomes possible to inherit the same name (e.g., function, typedef, etc.) from more than one base class. That leads to new opportunities for ambiguity. For example:

Click here to view code image

```
class BorrowableItem {                // something a library
lets you borrow
public:
  void checkOut();                    // check the item out from
the library

  ...
};

class ElectronicGadget {
private:
  bool checkOut() const;              // perform self-test,
return whether

  ...                                 // test succeeds
};

class MP3Player:                      // note MI here
  public BorrowableItem,              // (some libraries loan
MP3 players)
  public ElectronicGadget
{ ... };                                    // class definition is
unimportant
MP3Player mp;

mp.checkOut();                              // ambiguous! which
checkOut?
```

Note that in this example, the call to `checkOut` is ambiguous, even though only one of the two functions is accessible. (`checkOut` is public in `BorrowableItem` but private in `ElectronicGadget`.) That's in accord with the C++ rules for resolving calls to overloaded functions: before seeing whether a function is accessible, C++ first identifies the function that's the best match for the call. It checks accessibility only after finding the best-match function. In this case, the name `checkOut` is ambiguous during name lookup, so neither function overload resolution nor best match determination takes place. The accessibility of `ElectronicGadget::checkOut` is therefore never examined.

To resolve the ambiguity, you must specify which base class's function to call:

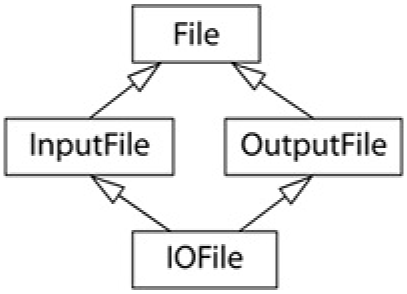
```
mp.BorrowableItem::checkOut();               // ah, that
checkOut...
```

You could try to explicitly call `ElectronicGadget::checkOut`, too, of course, but then the ambiguity error would be replaced with a “you're trying to call a private member function” error.

Multiple inheritance just means inheriting from more than one base class, but it is not uncommon for MI to be found in hierarchies that have higher-level base classes, too. That can lead to what is sometimes known as the “deadly MI diamond”

Click here to view code image

```
class File { ... };
class InputFile: public File { ... };
class OutputFile: public File { ... };
class IOFile: public InputFile,
              public OutputFile
{ ... };
```

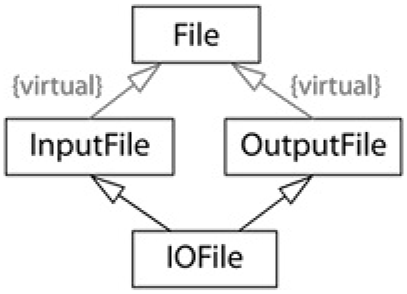


Any time you have an inheritance hierarchy with more than one path between a base class and a derived class (such as between `File` and `IOFile` above, which has paths through both `InputFile` and `OutputFile`), you must confront the question of whether you want the data members in the base class to be replicated for each of the paths. For example, suppose that the `File` class has a data member, `fileName`. How many copies of this field should `IOFile` have? On the one hand, it inherits a copy from each of its base classes, so that suggests that `IOFile` should have two `fileName` data members. On the other hand, simple logic says that an `IOFile` object has only one file name, so the `fileName` field it inherits through its two base classes should not be replicated.

C++ takes no position on this debate. It happily supports both options, though its default is to perform the replication. If that's not what you want, you must make the class with the data (i.e., `File`) a *virtual base class*. To do that, you have all classes that immediately inherit from it use *virtual inheritance*:

Click here to view code image

```
class File { ... };
class InputFile: virtual public File { ... };
class OutputFile: virtual public File { ... };
class IOFile: public InputFile,
              public OutputFile
{ ... };
```



The standard C++ library contains an MI hierarchy just like this one, except the classes are class templates, and the names are `basic_ios`, `basic_istream`, `basic_ostream`, and `basic_iostream` instead of `File`, `InputFile`, `OutputFile`, and `IOFile`.

From the viewpoint of correct behavior, public inheritance should always be virtual. If that were the only point of view, the rule would be simple: anytime you use public inheritance, use *virtual* public inheritance. Alas, correctness is not the only perspective. Avoiding the replication of inherited fields requires some behind-the-scenes legerdemain on the part of compilers, and the result is that objects created from classes using virtual inheritance are generally larger than they would be without virtual inheritance. Access to data members in virtual base classes is also slower than to those in non-virtual base classes. The details varies from compiler to compiler, but the basic thrust is clear: virtual inheritance costs.

It costs in other ways, too. The rules governing the initialization of virtual base classes are more complicated and less intuitive than are those for non-virtual bases. The responsibility for initializing a virtual base is borne by the *most derived class* in the hierarchy. Implications of this rule include (1) classes derived from virtual bases that require initialization must be aware of their virtual bases, no matter how far distant the bases are, and (2) when a new derived class is added to the hierarchy, it must assume initialization responsibilities for its virtual bases (both direct and indirect).

My advice on virtual base classes (i.e., on virtual inheritance) is simple. First, don't use virtual bases unless you need to. By default, use non-virtual inheritance. Second, if you must use virtual base classes, try to avoid putting data in them. That way you won't have to worry about oddities in the initialization (and, as it turns out, assignment) rules for such classes. It's worth noting that Interfaces in Java and .NET, which are in many ways comparable to virtual base classes in C++, are not allowed to contain any data.

Let us now turn to the following C++ Interface class (see Item 31) for modeling persons:

Click here to view code image

```
class IPerson {
public:
  virtual ~IPerson();
  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
};
```

`IPerson` clients must program in terms of `IPerson` pointers and references, because abstract classes cannot be instantiated. To create objects that can be manipulated as `IPerson` objects, clients of `IPerson` use factory functions (again, see Item 31) to instantiate concrete classes derived from `IPerson`:

Click here to view code image

```
// factory function to create a Person object from a unique
database ID;
// see Item 18 for why the return type isn't a raw pointer
std::tr1::shared_ptr<IPerson> makePerson(DatabaseID
personIdentifier);

// function to get a database ID from the user
DatabaseID askUserForDatabaseID();


DatabaseID id(askUserForDatabaseID());
std::tr1::shared_ptr<IPerson> pp(makePerson(id));    //
create an object
                                                      //
supporting the
                                                      //
IPerson interface

...                                                   //
manipulate *pp via
                                                      //
IPerson's member
                                                      //
functions
```

But how does `makePerson` create the objects to which it returns pointers? Clearly, there must be some concrete class derived from `IPerson` that `makePerson` can instantiate.

Suppose this class is called `CPerson`. As a concrete class, `CPerson` must provide implementations for the pure virtual functions it inherits from `IPerson`. It could write these from scratch, but it would be better to take advantage of existing components that do most or all of what's necessary. For example, suppose an old database-specific class `PersonInfo` offers the essence of what `CPerson` needs:

Click here to view code image

```
class PersonInfo {
public:
  explicit PersonInfo(DatabaseID pid);
  virtual ~PersonInfo();

  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  ...

private:
  virtual const char * valueDelimOpen() const;       // see
  virtual const char * valueDelimClose() const;      // below
  ...
};
```

You can tell this is an old class, because the member functions return `const char*`s instead of `string` objects. Still, if the shoe fits, why not wear it? The names of this class's member functions suggest that the result is likely to be pretty comfortable.

You come to discover that `PersonInfo` was designed to facilitate printing database fields in various formats, with the beginning and end of each field value delimited by special strings. By default, the opening and closing delimiters for field values are square brackets, so the field value “Ring-tailed Lemur” would be formatted this way:

```
[Ring-tailed Lemur]
```

In recognition of the fact that square brackets are not universally desired by clients of `PersonInfo`, the virtual functions `valueDelimOpen` and `valueDelimClose` allow derived classes to specify their own opening and closing delimiter strings. The implementations of `PersonInfo`'s member functions call these virtual functions to add the appropriate delimiters to the values they return. Using `PersonInfo::theName` as an example, the code looks like this:

Click here to view code image

```
const char * PersonInfo::valueDelimOpen() const
{
  return "[";                        // default opening
delimiter
}

const char * PersonInfo::valueDelimClose() const
{
  return "]";                        // default closing
delimiter
}

const char * PersonInfo::theName() const
{
  // reserve buffer for return value; because this is
  // static, it's automatically initialized to all zeros
  static char value[Max_Formatted_Field_Value_Length];

  // write opening delimiter
  std::strcpy(value, valueDelimOpen());

  append to the string in value this object's   name field
(being careful
  to avoid buffer overruns!)

  // write closing delimiter
  std::strcat(value, valueDelimClose());

  return value;
}
```

One might question the antiquated design of `PersonInfo::theName` (especially the use of a fixed-size static buffer, something that's rife for both overrun and threading problems — see also Item 21), but set such questions aside and focus instead on this: `theName` calls `valueDelimOpen` to generate the opening delimiter of the string it will return, then it generates the name value itself, then it calls `valueDelimClose`. Because `valueDelimOpen` and `valueDelimClose` are virtual functions, the result returned by `theName` is dependent not only on `PersonInfo` but also on the classes derived from `PersonInfo`.

As the implementer of `CPerson`, that's good news, because while perusing the fine print in the `IPerson` documentation, you discover that `name` and `birthDate` are required to return unadorned values, i.e., no delimiters are allowed. That is, if a person is named Homer, a call to that person's `name` function should return “Homer”, not “[Homer]”.

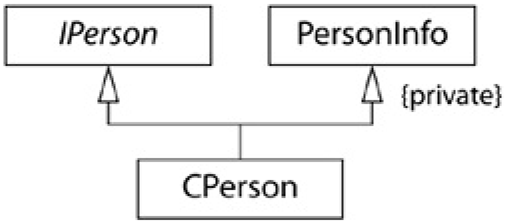
The relationship between `CPerson` and `PersonInfo` is that `PersonInfo` happens to have some functions that would make `CPerson` easier to implement. That's all. Their relationship is thus is-implemented-in-terms-of, and we know that can be represented in two ways: via composition (see Item 38) and via private inheritance (see Item 39). Item 39 points out that composition is the generally preferred approach, but inheritance is necessary if virtual functions are to be redefined. In this case, `CPerson` needs to redefine `valueDelimOpen` and `valueDelimClose`, so simple composition won't do. The most straightforward solution is to have `CPerson` privately inherit from `PersonInfo`, though Item 39 explains that with a bit more work, `CPerson` could also use a combination of composition and inheritance to effectively redefine `PersonInfo`'s virtuals. Here, we'll use private inheritance.

But `CPerson` must also implement the `IPerson` interface, and that calls for public inheritance. This leads to one reasonable application of multiple inheritance: combine public inheritance of an interface with private inheritance of an implementation:

Click here to view code image

```
class IPerson {                            // this class
specifies the
public:                                    // interface to be
implemented
  virtual ~IPerson();

  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
};

class DatabaseID { ... };                  // used below;
details are
                                           // unimportant

class PersonInfo {                         // this class has
functions
public:                                    // useful in
implementing
  explicit PersonInfo(DatabaseID pid);     // the IPerson
interface
  virtual ~PersonInfo();


...
private:
virtual const char * theName() const;
  virtual const char * theBirthDate() const;

  virtual const char * valueDelimOpen() const;
  virtual const char * valueDelimClose() const;
  ...
};

class CPerson: public IPerson, private PersonInfo {      //
note use of MI
public:
  explicit CPerson(    DatabaseID pid): PersonInfo(pid) {}
  virtual std::string name() const                       //
implementations
  { return PersonInfo::theName(); }                      // of
the required
                                                         //
IPerson member
  virtual std::string birthDate() const                  //
functions
  { return PersonInfo::theBirthDate(); }
private:                                                 //
redefinitions of
  const char * valueDelimOpen() const { return ""; }     //
inherited virtual
  const char * valueDelimClose() const { return ""; }    //
delimiter
};                                                       //
functions
```

In UML, the design looks like this:



This example demonstrates that MI can be both useful and comprehensible.

At the end of the day, multiple inheritance is just another tool in the object-oriented toolbox. Compared to single inheritance, it's typically more complicated to use and more complicated to understand, so if you've got an SI design that's more or less equivalent to an MI design, the SI design is almost certainly preferable. If the only design you can come up with involves MI, you should think a little harder — there's almost certainly *some* way to make SI work. At the same time, MI is sometimes the clearest, most maintainable, most reasonable way to get the job done. When that's the case, don't be afraid to use it. Just be sure to use it judiciously.

Things to Remember

- Multiple inheritance is more complex than single inheritance. It can lead to new ambiguity issues and to the need for virtual inheritance.
- Virtual inheritance imposes costs in size, speed, and complexity of initialization and assignment. It's most practical when virtual base classes have no data.
- Multiple inheritance does have legitimate uses. One scenario involves combining public inheritance from an Interface class with private inheritance from a class that helps with implementation.

7. Templates and Generic Programming

The initial motivation for C++ templates was straightforward: to make it possible to create type-safe containers like `vector`, `list`, and `map`. The more people worked with templates, however, the wider the variety of things they found they could do with them. Containers were good, but generic programming — the ability to write code that is independent of the types of objects being manipulated — was even better. STL algorithms like `for_each`, `find`, and `merge` are examples of such programming. Ultimately, it was discovered that the C++ template mechanism is itself Turing-complete: it can be used to compute any computable value. That led to template metaprogramming: the creation of programs that execute inside C++ compilers and that stop running when compilation is complete. These days, containers are but a small part of the C++ template pie. Despite the breadth of template applications, however, a set of core ideas underlie all template-based programming. Those ideas are the focus of this chapter.

This chapter won't make you an expert template programmer, but it will make you a better one. It will also give you information you need to expand your template-programming boundaries as far as you desire.

Item 41: Understand implicit interfaces and compile-time polymorphism

The world of object-oriented programming revolves around *explicit* interfaces and *runtime* polymorphism. For example, given this (meaningless) class,

Click here to view code image

```
class Widget {
public:
  Widget();
  virtual ~Widget();

  virtual std::size_t size() const;
  virtual void normalize();
  void swap(Widget& other);                 // see Item 25

  ...
};
```

and this (equally meaningless) function,

Click here to view code image

```
void doProcessing(Widget& w)
{
  if (w.size() > 10 && w != someNastyWidget) {
      Widget temp(w);
      temp.normalize();
      temp.swap(w);
  }
}
```

we can say this about `w` in `doProcessing`:

- Because `w` is declared to be of type `Widget`, `w` must support the `Widget` interface. We can look up this interface in the source code (e.g., the `.h` file for `Widget`) to see exactly what it looks like, so I call this an *explicit interface* — one explicitly visible in the source code.
- Because some of `Widget`'s member functions are virtual, `w`'s calls to those functions will exhibit *runtime polymorphism*: the specific function to call will be determined at runtime based on `w`'s dynamic type (see Item 37).

The world of templates and generic programming is fundamentally different. In that world, explicit interfaces and runtime polymorphism continue to exist, but they're less important. Instead, *implicit interfaces* and *compile-time polymorphism* move to the fore. To see how this is the case, look what happens when we turn `doProcessing` from a function into a function template:

Click here to view code image

```
template<typename T>
void doProcessing(T& w)
{
  if (w.size() > 10 && w != someNastyWidget) {
     T temp(w);
     temp.normalize();
     temp.swap(w);
  }
}
```

Now what can we say about `w` in `doProcessing`?

- The interface that `w` must support is determined by the operations performed on `w` in the template. In this example, it appears that `w`'s type (`T`) must support the `size`, `normalize`, and `swap` member functions; copy construction (to create `temp`); and comparison for inequality (for comparison with `someNastyWidget`). We'll soon see that this isn't quite accurate, but it's true enough for now. What's important is that the set of expressions that must be valid in order for the template to compile is the *implicit interface* that `T` must support.
- The calls to functions involving `w` such as `operator>` and `operator!=` may involve instantiating templates to make these calls succeed. Such instantiation occurs during compilation. Because instantiating function templates with different template parameters leads to different functions being called, this is known as *compile-time polymorphism*.

Even if you've never used templates, you should be familiar with the difference between runtime and compile-time polymorphism, because it's similar to the difference between the process of determining which of a set of overloaded functions should be called (which takes place during compilation) and dynamic binding of virtual function calls (which takes place at runtime). The difference between explicit and implicit interfaces is new to templates, however, and it bears closer examination.

An explicit interface typically consists of function signatures, i.e., function names, parameter types, return types, etc. The `Widget` class public interface, for example,

Click here to view code image

```
class Widget {
public:
  Widget();
  virtual ~Widget();

  virtual std::size_t size() const;
  virtual void normalize();
  void swap(Widget& other);
};
```

consists of a constructor, a destructor, and the functions `size`, `normalize`, and `swap`, along with the parameter types, return types, and constnesses of these functions. (It also includes the compiler-generated copy constructor and copy assignment operator — see Item 5.) It could also include typedefs and, if you were so bold as to violate Item 22's advice to make data members private, data members, though in this case, it does not.

An implicit interface is quite different. It is not based on function signatures. Rather, it consists of valid *expressions*. Look again at the conditional at the beginning of the `doProcessing` template:

Click here to view code image

```
template<typename T>
void doProcessing(T& w)
{
  if (w.size() > 10 && w != someNastyWidget) {
  ...
```

The implicit interface for `T` (`w`'s type) appears to have these constraints:

- It must offer a member function named `size` that returns an integral value.
- It must support an `operator!=` function that compares two objects of type `T`. (Here, we assume that `someNastyWidget` is of type `T`.)

Thanks to the possibility of operator overloading, neither of these constraints need be satisfied. Yes, `T` must support a `size` member function, though it's worth mentioning that the function might be inherited from a base class. But this member function need not return an integral type. It need not even return a numeric type. For that matter, it need not even return a type for which `operator>` is defined! All it needs to do is return an object of some type `X` such that there is an `operator>` that can be called with an object of type `X` and an `int` (because 10 is of type `int`). The `operator>` need not take a parameter of type `X`, because it could take a parameter of type `Y`, and that would be okay as long as there were an implicit conversion from objects of type `X` to objects of type `Y`!

Similarly, there is no requirement that `T` support `operator!=`, because it would be just as acceptable for `operator!=` to take one object of type `X` and one object of type `Y`. As long as `T` can be converted to `X` and `someNastyWidget`'s type can be converted to `Y`, the call to `operator!=` would be valid.

(As an aside, this analysis doesn't take into account the possibility that `operator&&` could be overloaded, thus changing the meaning of the above expression from a conjunction to something potentially quite different.)

Most people's heads hurt when they first start thinking about implicit interfaces this way, but there's really no need for aspirin. Implicit interfaces are simply made up of a set of valid expressions. The expressions themselves may look complicated, but the constraints they impose are generally straightforward. For example, given the conditional,

```
if (w.size() > 10 && w != someNastyWidget) ...
```

it's hard to say much about the constraints on the functions `size`, `operator>`, `operator&&`, or `operator!=`, but it's easy to identify the constraint on the expression as a whole. The conditional part of an `if` statement must be a boolean expression, so regardless of the exact types involved, whatever "`w.size() > 10 && w != someNastyWidget`" yields, it must be compatible with `bool`. This is part of the implicit interface the template `doProcessing` imposes on its type parameter `T`. The rest of the interface required by `doProcessing` is that calls to the copy constructor, to `normalize`, and to `swap` must be valid for objects of type `T`.

The implicit interfaces imposed on a template's parameters are just as real as the explicit interfaces imposed on a class's objects, and both are checked during compilation. Just as you can't use an object in a way contradictory to the explicit interface its class offers (the code won't compile), you can't try to use an object in a template unless that object supports the implicit interface the template requires (again, the code won't compile).

Things to Remember

- Both classes and templates support interfaces and polymorphism.
- For classes, interfaces are explicit and centered on function signatures. Polymorphism occurs at runtime through virtual functions.
- For template parameters, interfaces are implicit and based on valid expressions. Polymorphism occurs during compilation through template instantiation and function overloading resolution.

Item 42: Understand the two meanings of `typename`

Question: what is the difference between `class` and `typename` in the following template declarations?

Click here to view code image

```
template<class T> class Widget;                 // uses
"class"

template<typename T> class Widget;              // uses
"typename"
```

Answer: nothing. When declaring a template type parameter, `class` and `typename` mean exactly the same thing. Some programmers prefer `class` all the time, because it's easier to type. Others (including me) prefer `typename`, because it suggests that the parameter need not be a class type. A few developers employ `typename` when any type is allowed and reserve `class` for when only user-defined types are acceptable. But from C++'s point of view, `class` and `typename` mean exactly the same thing when declaring a template parameter.

C++ doesn't always view `class` and `typename` as equivalent, however. Sometimes you must use `typename`. To understand when, we have to talk about two kinds of names you can refer to in a template.

Suppose we have a template for a function that takes an STL-compatible container holding objects that can be assigned to `int`s. Further suppose that this function simply prints the value of its second element. It's a silly function implemented in a silly way, and as I've written it below, it shouldn't even compile, but please overlook those things — there's a method to my madness:

Click here to view code image

```
template<typename C>                            // print 2nd
element in
void print2nd(const C& container)               // container;
{                                               // this is
not valid C++!
  if (container.size() >= 2) {
     C::const_iterator iter(container.begin()); // get
iterator to 1st element
     ++iter;                                    // move iter
to 2nd element
     int value = *iter;                         // copy that
element to an int
     std::cout << value;                        // print the
int
  }
}
```

I've highlighted the two local variables in this function, `iter` and `value`. The type of `iter` is `C::const_iterator`, a type that depends on the template parameter `C`. Names in a template that are dependent on a template parameter are called *dependent names*. When a dependent name is nested inside a class, I call it a *nested dependent name*. `C::const_iterator` is a nested dependent name. In fact, it's a *nested dependent type name*, i.e., a nested dependent name that refers to a type.

The other local variable in `print2nd`, `value`, has type `int`. `int` is a name that does not depend on any template parameter. Such names are known as *non-dependent names*, (I have no idea why they're not called independent names. If, like me, you find the term “non-dependent” an abomination, you have my sympathies, but “non-dependent” is the term for these kinds of names, so, like me, roll your eyes and resign yourself to it.)

Nested dependent names can lead to parsing difficulties. For example, suppose we made `print2nd` even sillier by starting it this way:

```
template<typename C>
void print2nd(const C& container)
{
  C::const_iterator * x;
  ...
}
```

This looks like we're declaring `x` as a local variable that's a pointer to a `C::const_iterator`. But it looks that way only because we “know” that `C::const_iterator` is a type. But what if `C::const_iterator` weren't a type? What if `C` had a static data member that happened to be named `const_iterator`, and what if `x` happened to be the name of a global variable? In that case, the code above wouldn't declare a local variable, it would be a multiplication of `C::const_iterator` by `x`! Sure, that sounds crazy, but it's *possible*, and people who write C++ parsers have to worry about all possible inputs, even the crazy ones.

Until `C` is known, there's no way to know whether `C::const_iterator` is a type or isn't, and when the template `print2nd` is parsed, `C` isn't known. C++ has a rule to resolve this ambiguity: if the parser encounters a nested dependent name in a template, it assumes that the name is *not* a type unless you tell it otherwise. By default, nested dependent names are *not* types. (There is an exception to this rule that I'll get to in a moment.)

With that in mind, look again at the beginning of `print2nd`:

Click here to view code image

```
template<typename C>
void print2nd(const C& container)
{
  if (container.size() >= 2) {
     C::const_iterator iter(container.begin());   // this
name is assumed to
     ...                                          // not be a
type
```

Now it should be clear why this isn't valid C++. The declaration of `iter` makes sense only if `C::const_iterator` is a type, but we haven't told C++ that it is, and C++ assumes that it's not. To rectify the situation, we have to tell C++ that `C::const_iterator` is a type. We do that by putting `typename` immediately in front of it:

Click here to view code image

```
template<typename C>                           // this is
valid C++
void print2nd(const C& container)
{
  if (container.size() >= 2) {
    typename C::const_iterator iter(container.begin());
    ...
  }
}
```

The general rule is simple: anytime you refer to a nested dependent type name in a template, you must immediately precede it by the word `typename`. (Again, I'll describe an exception shortly.)

`typename` should be used to identify only nested dependent type names; other names shouldn't have it. For example, here's a function template that takes both a container and an iterator into that container:

[Click here to view code image](#)

```
template<typename C>                    // typename allowed
(as is "class")
void f(const C& container,              // typename not
allowed
     typename C::iterator iter);        // typename required
```

`C` is not a nested dependent type name (it's not nested inside anything dependent on a template parameter), so it must not be preceded by `typename` when declaring `container`, but `C::iterator` is a nested dependent type name, so it's required to be preceded by `typename`.

The exception to the "`typename` must precede nested dependent type names" rule is that `typename` must not precede nested dependent type names in a list of base classes or as a base class identifier in a member initialization list. For example:

[Click here to view code image](#)

```
template<typename T>
class Derived: public Base<T>::Nested { // base class list:
typename not
public:                                 // allowed
  explicit Derived(int x)
  : Base<T>::Nested(x)                  // base class
identifier in mem.
  {                                     // init. list:
typename not allowed

    typename Base<T>::Nested temp;      // use of nested
dependent type
    ...                                 // name not in a base
class list or
  }                                     // as a base class
identifier in a
  ...                                   // mem. init. list:
typename required
};
```

Such inconsistency is irksome, but once you have a bit of experience under your belt, you'll barely notice it.

Let's look at one last `typename` example, because it's representative of something you're going to see in real code. Suppose we're writing a function template that takes an iterator, and we want to make a local copy, `temp`, of the object the iterator points to. We can do it like this:

Click here to view code image

```
template<typename IterT>
void workWithIterator(IterT iter)
{
  typename std::iterator_traits<IterT>::value_type
temp(*iter);
  ...
}
```

Don't let the `std::iterator_traits<IterT>::value_type` startle you. That's just a use of a standard traits class (see Item 47), the C++ way of saying "the type of thing pointed to by objects of type `IterT`." The statement declares a local variable (`temp`) of the same type as what `IterT` objects point to, and it initializes `temp` with the object that `iter` points to. If `IterT` is `vector<int>::iterator`, `temp` is of type `int`. If `IterT` is `list<string>::iterator`, `temp` is of type `string`. Because `std::iterator_traits<IterT>::value_type` is a nested dependent type name (`value_type` is nested inside `iterator_traits<IterT>`, and `IterT` is a template parameter), we must precede it by `typename`.

If you think reading `std::iterator_traits<IterT>::value_type` is unpleasant, imagine what it's like to type it. If you're like most programmers, the thought of typing it more than once is ghastly, so you'll want to create a typedef. For traits member names like `value_type` (again, see Item 47 for information on traits), a common convention is for the typedef name to be the same as the traits member name, so such a local typedef is often defined like this:

Click here to view code image

```
template<typename IterT>
void workWithIterator(IterT iter)
{
  typedef typename std::iterator_traits<IterT>::value_type
value_type;

  value_type temp(*iter);
  ...
}
```

Many programmers find the “`typedef typename`” juxtaposition initially jarring, but it's a logical fallout from the rules for referring to nested dependent type names. You'll get used to it fairly quickly. After all, you have strong motivation. How many times do you want to type `typename std::iterator_traits<IterT>::value_type`?

As a closing note, I should mention that enforcement of the rules surrounding `typename` vary from compiler to compiler. Some compilers accept code where `typename` is required but missing; some accept code where `typename` is present but not allowed; and a few (usually older ones) reject `typename` where it's present and required. This means that the interaction of `typename` and nested dependent type names can lead to some mild portability headaches.

Things to Remember

- When declaring template parameters, `class` and `typename` are interchangeable.
- Use `typename` to identify nested dependent type names, except in base class lists or as a base class identifier in a member initialization list.

Item 43: Know how to access names in templatized base classes

Suppose we need to write an application that can send messages to several different companies. Messages can be sent in either encrypted or cleartext (unencrypted) form. If we have enough information during compilation to determine which messages will go to which companies, we can employ a template-based solution:

Click here to view code image

```
class CompanyA {
public:
  ...
  void sendCleartext(const std::string& msg);
  void sendEncrypted(const std::string& msg);
  ...
};
```

```
class CompanyB {
public:
  ...
  void sendCleartext(const std::string& msg);
  void sendEncrypted(const std::string& msg);
  ...
};
...                                     // classes for other
companies

class MsgInfo { ... };                  // class for holding
information
                                        // used to create a
message
template<typename Company>
class MsgSender {
public:
  ...                                   // ctors, dtor, etc.

  void sendClear(const MsgInfo& info)
  {
    std::string msg;
    create msg from info;

    Company c;
    c.sendCleartext(msg);
  }

  void sendSecret(const MsgInfo& info)   // similar to
sendClear, except
  { ... }                                // calls
c.sendEncrypted
};
```

This will work fine, but suppose we sometimes want to log some information each time we send a message. A derived class can easily add that capability, and this seems like a reasonable way to do it:

Click here to view code image

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...                                    // ctors, dtor, etc.
  void sendClearMsg(const MsgInfo& info)
  {
    write "before sending" info to the log;

    sendClear(info);                     // call base class
function;
                                         // this code will
not compile!
    write "after sending" info to the log;
  }
  ...

};
```

Note how the message-sending function in the derived class has a different name (`sendClearMsg`) from the one in its base class (there, it's called `sendClear`). That's good design, because it side-steps the issue of hiding inherited names (see Item 33) as well as the problems inherent in redefining an inherited non-virtual function (see Item 36). But the code above won't compile, at least not with conformant compilers. Such compilers will complain that `sendClear` doesn't exist. We can see that `sendClear` is in the base class, but compilers won't look for it there. We need to understand why.

The problem is that when compilers encounter the definition for the class template `LoggingMsgSender`, they don't know what class it inherits from. Sure, it's `MsgSender<Company>`, but `Company` is a template parameter, one that won't be known until later (when `LoggingMsgSender` is instantiated). Without knowing what `Company` is, there's no way to know what the class `MsgSender<Company>` looks like. In particular, there's no way to know if it has a `sendClear` function.

To make the problem concrete, suppose we have a class `CompanyZ` that insists on encrypted communications:

Click here to view code image

```
class CompanyZ {                             // this class
offers no
public:                                      // sendCleartext
function
  ...
  void sendEncrypted(const std::string& msg);
  ...
};
```

The general `MsgSender` template is inappropriate for `CompanyZ`, because that template offers a `sendClear` function that makes no sense for `CompanyZ` objects. To rectify that problem, we can create a specialized version of `MsgSender` for `CompanyZ`:

[Click here to view code image](#)

```
template<>                                 // a total
specialization of
class MsgSender<CompanyZ> {                // MsgSender; the
same as the
public:                                    // general
template, except
  ...                                      // sendClear is
omitted
  void sendSecret(const MsgInfo& info)
  { ... }
};
```

Note the “`template <>`” syntax at the beginning of this class definition. It signifies that this is neither a template nor a standalone class. Rather, it's a specialized version of the `MsgSender` template to be used when the template argument is `CompanyZ`. This is known as a *total template specialization*: the template `MsgSender` is specialized for the type `CompanyZ`, and the specialization is *total* — once the type parameter has been defined to be `CompanyZ`, no other aspect of the template's parameters can vary.

Given that `MsgSender` has been specialized for `CompanyZ`, consider again the derived class `LoggingMsgSender`:

[Click here to view code image](#)

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...
  void sendClearMsg(const MsgInfo& info)
  {
    write "before sending" info to the log;

    sendClear(info);                          // if Company
== CompanyZ,
                                              // this
function doesn't exist!
    write "after sending" info to the log;
  }
  ...

};
```

As the comment notes, this code makes no sense when the base class is `MsgSender<CompanyZ>`, because that class offers no `sendClear` function. That's why C++ rejects the call: it recognizes that base class templates may be specialized and that such specializations may not offer the same interface as the general template. As a result, it generally refuses to look in templatized base classes for inherited names. In some sense, when we cross from Object-Oriented C++ to Template C++ (see Item 1), inheritance stops working.

To restart it, we have to somehow disable C++'s “don't look in templatized base classes” behavior. There are three ways to do this. First, you can preface calls to base class functions with “`this->`”:

Click here to view code image

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:

  ...

  void sendClearMsg(const MsgInfo& info)
  {
    write "before sending" info to the log;

    this->sendClear(info);                // okay, assumes
that
                                          // sendClear will
be inherited
    write "after sending" info to the log;
  }

  ...

};
```

Second, you can employ a `using` declaration, a solution that should strike you as familiar if you've read Item 33. That Item explains how `using` declarations bring hidden base class names into a derived class's scope. We can therefore write `sendClearMsg` like this:

Click here to view code image

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  using MsgSender<Company>::sendClear;   // tell compilers to
assume
  ...                                    // that sendClear is
in the
                                         // base class
  void sendClearMsg(const MsgInfo& info)
  {
    ...
    sendClear(info);                   // okay, assumes that
    ...                                // sendClear will be
inherited
  }

  ...
};
```

(Although a `using` declaration will work both here and in Item 33, the problems being solved are different. Here, the situation isn't that base class names are hidden by derived class names, it's that compilers don't search base class scopes unless we tell them to.)

A final way to get your code to compile is to explicitly specify that the function being called is in the base class:

Click here to view code image

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...
  void sendClearMsg(const MsgInfo& info)
  {
    ...
    MsgSender<Company>::sendClear(info);      // okay,
assumes that
    ...                                       // sendClear
will be
  }                                            // inherited

  ...
};
```

This is generally the least desirable way to solve the problem, because if the function being called is virtual, explicit qualification turns off the virtual binding behavior.

From a name visibility point of view, each of these approaches does the same thing: it promises compilers that any subsequent specializations of the base class template will support the interface offered by the general template. Such a promise is all compilers need when they parse a derived class template like `LoggingMsgSender`, but if the promise turns out to be unfounded, the truth will emerge during subsequent compilation. For example, if the source code later contains this,

Click here to view code image

```
LoggingMsgSender<CompanyZ> zMsgSender;

MsgInfo msgData;

...                                           // put info in
msgData

zMsgSender.sendClearMsg(msgData);             // error! won't
compile
```

the call to `sendClearMsg` won't compile, because at this point, compilers know that the base class is the template specialization `MsgSender<CompanyZ>`, and they know that class doesn't offer the `sendClear` function that `sendClearMsg` is trying to call.

Fundamentally, the issue is whether compilers will diagnose invalid references to base class members sooner (when derived class template definitions are parsed) or later (when those templates are instantiated with specific template arguments). C++'s policy is to prefer early diagnoses, and that's why it assumes it knows nothing about the contents of base classes when those classes are instantiated from templates.

Things to Remember

- In derived class templates, refer to names in base class templates via a “`this->`” prefix, via `using` declarations, or via an explicit base class qualification.

Item 44: Factor parameter-independent code out of templates

Templates are a wonderful way to save time and avoid code replication. Instead of typing 20 similar classes, each with 15 member functions, you type one class template, and you let compilers instantiate the 20 specific classes and 300 functions you need. (Member functions of class templates are implicitly instantiated only when used, so you should get the full 300 member functions only if each is actually used.) Function templates are similarly appealing. Instead of writing many functions, you write one function template and let the compilers do the rest. Ain't technology grand?

Yes, well...sometimes. If you're not careful, using templates can lead to *code bloat*: binaries with replicated (or almost replicated) code, data, or both. The result can be source code that looks fit and trim, yet object code that's fat and flabby. Fat and flabby is rarely fashionable, so you need to know how to avoid such binary bombast.

Your primary tool has the imposing name *commonality and variability analysis*, but there's nothing imposing about the idea. Even if you've never written a template in your life, you do such analysis all the time.

When you're writing a function and you realize that some part of the function's implementation is essentially the same as another function's implementation, do you just replicate the code? Of course not. You factor the common code out of the two functions, put it into a third function, and have both of the other functions call the new one. That is, you analyze the two functions to find the parts that are common and those that vary, you move the common parts into a new function, and you keep the varying parts in the original functions. Similarly, if you're writing a class and you realize that some parts of the class are the same as parts of another class, you don't replicate the common parts. Instead, you move the common parts to a new class, then you use inheritance or composition (see Items 32, 38, and 39) to give the original classes access to the common features. The parts of the original classes that differ — the varying parts — remain in their original locations.

When writing templates, you do the same analysis, and you avoid replication in the same ways, but there's a twist. In non-template code, replication is explicit: you can *see* that there's duplication between two functions or two classes. In template code, replication is implicit: there's only one copy of the template source code, so you have to train yourself to sense the replication that may take place when a template is instantiated multiple times.

For example, suppose you'd like to write a template for fixed-size square matrices that, among other things, support matrix inversion.

Click here to view code image

```
template<typename T,           // template for n x n matrices
of
         std::size_t n>        // objects of type T; see
below for info
class SquareMatrix {           // on the size_t parameter
public:
  ...
  void invert();              // invert the matrix in place
};
```

This template takes a type parameter, `T`, but it also takes a parameter of type `size_t` — a *non-type parameter*. Non-type parameters are less common than type parameters, but they're completely legal, and, as in this example, they can be quite natural.

Now consider this code:

Click here to view code image

```
SquareMatrix<double, 5> sm1;
...
sm1.invert();                  // call SquareMatrix<double,
5>::invert

SquareMatrix<double, 10> sm2;
...
sm2.invert();                  // call SquareMatrix<double,
10>::invert
```

Two copies of `invert` will be instantiated here. The functions won't be identical, because one will work on 5×5 matrices and one will work on 10 × 10 matrices, but other than the constants 5 and 10, the two functions will be the same. This is a classic way for template-induced code bloat to arise.

What would you do if you saw two functions that were character-for-character identical except for the use of 5 in one version and 10 in the other? Your instinct would be to create a version of the function that took a value as a parameter, then call the parameterized function with 5 or 10 instead of replicating the code. Your instinct serves you well! Here's a first pass at doing that for `SquareMatrix`:

Click here to view code image

```
template<typename T>                   // size-independent
base class for
class SquareMatrixBase {               // square matrices
protected:
  ...
  void invert(std::size_t matrixSize); // invert matrix of
the given size
  ...
};

template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
private:
  using SquareMatrixBase<T>::invert;   // make base class
version of invert
                                       // visible in this
class; see Items 33
                                       // and 43
public:
  ...
  void invert() { this->invert(n); }   // make inline call to
base class
};                                     // version of invert
```

As you can see, the parameterized version of `invert` is in a base class, `SquareMatrixBase`. Like `SquareMatrix`, `SquareMatrixBase` is a template, but unlike `SquareMatrix`, it's templatized only on the type of objects in the matrix, not on the size of the matrix. Hence, all matrices holding a given type of object will share a single `SquareMatrixBase` class. They will thus share a single copy of that class's version of `invert`.

`SquareMatrixBase::invert` is intended only to be a way for derived classes to avoid code replication, so it's `protected` instead of being `public`. The additional cost of calling it should be zero, because derived classes' `invert`s call the base class version using inline functions. (The `inline` is implicit — see Item 30.) Notice also that the inheritance between `SquareMatrix` and `SquareMatrixBase` is `private`. This accurately reflects the fact that the reason for the base class is only to facilitate the derived classes' implementations, not to express a conceptual is-a relationship between `SquareMatrix` and `SquareMatrixBase`. (For information on private inheritance, see Item 39.)

So far, so good, but there's a sticky issue we haven't addressed yet. How does `SquareMatrixBase::invert` know what data to operate on? It knows the size of the matrix from its parameter, but how does it know where the data for a particular matrix is? Presumably only the derived class knows that. How does the derived class communicate that to the base class so that the base class can do the inversion?

One possibility would be to add another parameter to `SquareMatrixBase::invert`, perhaps a pointer to the beginning of a chunk of memory with the matrix's data in it. That would work, but in all likelihood, `invert` is not the only function in `SquareMatrix` that can be written in a size-independent manner and moved into `SquareMatrixBase`. If there are several such functions, all will need a way to find the memory holding the values in the matrix. We could add an extra parameter to all of them, but we'd be telling `SquareMatrixBase` the same information repeatedly. That seems wrong.

An alternative is to have `SquareMatrixBase` store a pointer to the memory for the matrix values. And as long as it's storing that, it might as well store the matrix size, too. The resulting design looks like this:

Click here to view code image

```
template<typename T>
class SquareMatrixBase {
protected:
  SquareMatrixBase(std::size_t n, T *pMem)     // store
matrix size and a
  : size(n), pData(pMem) {}                    // ptr to
matrix values

  void setDataPtr(T *ptr) { pData = ptr; }     // reassign
pData
  ...

private:
  std::size_t size;                           // size of
matrix

  T *pData;                                   // pointer to
matrix values
};
```

This lets derived classes decide how to allocate the memory. Some implementations might decide to store the matrix data right inside the `SquareMatrix` object:

Click here to view code image

```
template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
public:
  SquareMatrix()                             // send matrix
size and
  : SquareMatrixBase<T>(n, data) {}          // data ptr to
base class
  ...

private:
  T data[n*n];
};
```

Objects of such types have no need for dynamic memory allocation, but the objects themselves could be very large. An alternative would be to put the data for each matrix on the heap:

Click here to view code image

```
template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
public:
  SquareMatrix()                          // set base class
data ptr to null,
  : SquareMatrixBase<T>(n, 0),            // allocate memory
for matrix
    pData(new T[n*n])                     // values, save a
ptr to the
```

```
  { this->setDataPtr(pData.get()); }          // memory, and give
a copy of it
  ...                                         // to the base
class

private:
  boost::scoped_array<T> pData;               // see Item 13 for
info on
};                                            //
boost::scoped_array
```

Regardless of where the data is stored, the key result from a bloat point of view is that now many — maybe all — of `SquareMatrix`'s member functions can be simple inline calls to (non-inline) base class versions that are shared with all other matrices holding the same type of data, regardless of their size. At the same time, `SquareMatrix` objects of different sizes are distinct types, so even though, e.g., `SquareMatrix<double, 5>` and `SquareMatrix<double, 10>` objects use the same member functions in `SquareMatrixBase<double>`, there's no chance of passing a `SquareMatrix<double, 5> object` to a function expecting a `SquareMatrix<double, 10>`. Nice, no?

Nice, yes, but not free. The versions of `invert` with the matrix sizes hardwired into them are likely to generate better code than the shared version where the size is passed as a function parameter or is stored in the object. For example, in the size-specific versions, the sizes would be compile-time constants, hence eligible for such optimizations as constant propagation, including their being folded into the generated instructions as immediate operands. That can't be done in the size-independent version.

On the other hand, having only one version of `invert` for multiple matrix sizes decreases the size of the executable, and that could reduce the program's working set size and improve locality of reference in the instruction cache. Those things could make the program run faster, more than compensating for any lost optimizations in size-specific versions of `invert`. Which effect would dominate? The only way to know is to try it both ways and observe the behavior on your particular platform and on representative data sets.

Another efficiency consideration concerns the sizes of objects. If you're not careful, moving size-independent versions of functions up into a base class can increase the overall size of each object. For example, in the code I just showed, each `SquareMatrix` object has a pointer to its data in the

`SquareMatrixBase` class, even though each derived class already has a way to get to the data. This increases the size of each `SquareMatrix` object by at least the size of a pointer. It's possible to modify the design so that these pointers are unnecessary, but, again, there are trade-offs. For example, having the base class store a `protected` pointer to the matrix data leads to the loss of encapsulation described in Item 22. It can also lead to resource management complications: if the base class stores a pointer to the matrix data, but that data may have been either dynamically allocated or physically stored inside the derived class object (as we saw), how will it be determined whether the pointer should be deleted? Such questions have answers, but the more sophisticated you try to be about them, the more complicated things become. At some point, a little code replication begins to look like a mercy.

This Item has discussed only bloat due to non-type template parameters, but type parameters can lead to bloat, too. For example, on many platforms, `int` and `long` have the same binary representation, so the member functions for, say, `vector<int>` and `vector<long>` would likely be identical — the very definition of bloat. Some linkers will merge identical function implementations, but some will not, and that means that some templates instantiated on both `int` and `long` could cause code bloat in some environments. Similarly, on most platforms, all pointer types have the same binary representation, so templates holding pointer types (e.g., `list<int*>`, `list<const int*>`, `list<SquareMatrix<long, 3>*>`, etc.) should often be able to use a single underlying implementation for each member function. Typically, this means implementing member functions that work with strongly typed pointers (i.e., `T*` pointers) by having them call functions that work with untyped pointers (i.e., `void*` pointers). Some implementations of the standard C++ library do this for templates like `vector`, `deque`, and `list`. If you're concerned about code bloat arising in your templates, you'll probably want to develop templates that do the same thing.

Things to Remember

- Templates generate multiple classes and multiple functions, so any template code not dependent on a template parameter causes bloat.
- Bloat due to non-type template parameters can often be eliminated by replacing template parameters with function

parameters or class data members.

- Bloat due to type parameters can be reduced by sharing implementations for instantiation types with identical binary representations.

Item 45: Use member function templates to accept “all compatible types.”

Smart pointers are objects that act much like pointers but add functionality pointers don't provide. For example, Item 13 explains how the standard `auto_ptr` and `tr1::shared_ptr` can be used to automatically delete heap-based resources at the right time. Iterators into STL containers are almost always smart pointers; certainly you couldn't expect to move a built-in pointer from one node in a linked list to the next by using “`++`,” yet that works for `list::iterator`s.

One of the things that real pointers do well is support implicit conversions. Derived class pointers implicitly convert into base class pointers, pointers to non-`const` objects convert into pointers to `const` objects, etc. For example, consider some conversions that can occur in a three-level hierarchy:

Click here to view code image

```
class Top { ... };
class Middle: public Top { ... };
class Bottom: public Middle { ... };
Top *pt1 = new Middle;                  // convert Middle*
⇒Top*
Top *pt2 = new Bottom;                  // convert Bottom*
⇒Top*
const Top *pct2 = pt1;                  // convert Top* ⇒
const Top*
```

Emulating such conversions in user-defined smart pointer classes is tricky. We'd need the following code to compile:

Click here to view code image

```
template<typename T>
class SmartPtr {
public:                            // smart pointers are
typically
  explicit SmartPtr(T *realPtr);    // initialized by built-
in pointers
  ...
};

SmartPtr<Top> pt1 =                 // convert
SmartPtr<Middle> ⇒
  SmartPtr<Middle>(new Middle);     //   SmartPtr<Top>

SmartPtr<Top> pt2 =                 // convert
SmartPtr<Bottom> ⇒
  SmartPtr<Bottom>(new Bottom);     //   SmartPtr<Top>

SmartPtr<const Top> pct2 = pt1;     // convert SmartPtr<Top>
⇒
                                    //  SmartPtr<const Top>
```

There is no inherent relationship among different instantiations of the same template, so compilers view `SmartPtr<Middle>` and `SmartPtr<Top>` as completely different classes, no more closely related than, say, `vector<float>` and `Widget`. To get the conversions among `SmartPtr` classes that we want, we have to program them explicitly.

In the smart pointer sample code above, each statement creates a new smart pointer object, so for now we'll focus on how to write smart pointer constructors that behave the way we want. A key observation is that there is no way to write out all the constructors we need. In the hierarchy above, we can construct a `SmartPtr<Top>` from a `SmartPtr<Middle>` or a `SmartPtr<Bottom>`, but if the hierarchy is extended in the future, `SmartPtr<Top>` objects will have to be constructible from other smart pointer types. For example, if we later add

```
class BelowBottom: public Bottom { ... };
```

we'll need to support the creation of `SmartPtr<Top>` objects from `SmartPtr<BelowBottom>` objects, and we certainly won't want to have to modify the `SmartPtr` template to do it.

In principle, the number of constructors we need is unlimited. Since a template can be instantiated to generate an unlimited number of functions, it seems that we don't need a constructor *function* for `SmartPtr`, we need a constructor *template*. Such templates are examples of *member function* *templates* (often just known as *member templates*) — templates that generate member functions of a class:

Click here to view code image

```
template<typename T>
class SmartPtr {
public:
  template<typename U>                       // member
template
  SmartPtr(const SmartPtr<U>& other);        // for a
"generalized
  ...                                        // copy
constructor"
};
```

This says that for every type `T` and every type `U`, a `SmartPtr<T>` can be created from a `SmartPtr<U>`, because `SmartPtr<T>` has a constructor that takes a `SmartPtr<U>` parameter. Constructors like this — ones that create one object from another object whose type is a different instantiation of the same template (e.g., create a `SmartPtr<T>` from a `SmartPtr<U>`) — are sometimes known as *generalized copy constructors*.

The generalized copy constructor above is not declared `explicit`. That's deliberate. Type conversions among built-in pointer types (e.g., from derived to base class pointers) are implicit and require no cast, so it's reasonable for smart pointers to emulate that behavior. Omitting `explicit` on the templatized constructor does just that.

As declared, the generalized copy constructor for `SmartPtr` offers more than we want. Yes, we want to be able to create a `SmartPtr<Top>` from a `SmartPtr<Bottom>`, but we don't want to be able to create a `SmartPtr<Bottom>` from a `SmartPtr<Top>`, as that's contrary to the meaning of public inheritance (see Item 32). We also don't want to be able to create a `SmartPtr<int>` from a `SmartPtr<double>`, because there is no corresponding implicit conversion from `int*` to `double*`. Somehow, we have to cull the herd of member functions that this member template will generate.

Assuming that `SmartPtr` follows the lead of `auto_ptr` and `tr1::shared_ptr` by offering a `get` member function that returns a copy of the built-in pointer held by the smart pointer object (see Item 15), we can use the implementation of the constructor template to restrict the conversions to those we want:

[Click here to view code image]

```
template<typename T>
class SmartPtr {
public:
  template<typename U>
  SmartPtr(const SmartPtr<U>& other)          // initialize
this held ptr
  : heldPtr(other.get()) { ... }              // with other's
held ptr

  T* get() const { return heldPtr; }
  ...

private:                                      // built-in
pointer held
  T *heldPtr;                                 // by the
SmartPtr
};
```

We use the member initialization list to initialize `SmartPtr<T>`'s data member of type `T*` with the pointer of type `U*` held by the `SmartPtr<U>`. This will compile only if there is an implicit conversion from a `U*` pointer to a `T*` pointer, and that's precisely what we want. The net effect is that `SmartPtr<T>` now has a generalized copy constructor that will compile only if passed a parameter of a compatible type.

The utility of member function templates isn't limited to constructors. Another common role for them is in support for assignment. For example, TR1's `shared_ptr` (again, see Item 13) supports construction from all compatible built-in pointers, `tr1::shared_ptr`s, `auto_ptr`s, and `tr1::weak_ptr`s (see Item 54), as well as assignment from all of those except `tr1::weak_ptr`s. Here's an excerpt from TR1's specification for `tr1::shared_ptr`, including its penchant for using `class` instead of `typename` when declaring template parameters. (As Item 42 explains, they mean exactly the same thing in this context.)

[Click here to view code image]

```
template<class T> class shared_ptr {
public:
  template<class Y>                                      //
construct from
    explicit shared_ptr(Y * p);                         //
any compatible
  template<class Y>                                     //
built-in pointer,
    shared_ptr(shared_ptr<Y> const& r);                 //
shared_ptr,
  template<class Y>                                     //
weak_ptr, or
    explicit shared_ptr(weak_ptr<Y> const& r);          //
auto_ptr
  template<class Y>
    explicit shared_ptr(auto_ptr<Y>& r);

  template<class Y>                                     //
assign from
    shared_ptr& operator=(shared_ptr<Y> const& r);      //
any compatible
  template<class Y>                                     //
shared_ptr or
    shared_ptr& operator=(auto_ptr<Y>& r);              //
auto_ptr
  ...
};
```

All these constructors are `explicit`, except the generalized copy constructor. That means that implicit conversion from one type of `shared_ptr` to another is allowed, but *implicit* conversion from a built-in pointer or other smart pointer type is not permitted. (*Explicit* conversion — e.g., via a cast — is okay.) Also interesting is how the `auto_ptr`s passed to `tr1::shared_ptr` constructors and assignment operators aren't declared `const`, in contrast to how the `tr1::shared_ptr`s and `tr1::weak_ptr`s are passed. That's a consequence of the fact that `auto_ptr`s stand alone in being modified when they're copied (see Item 13).

Member function templates are wonderful things, but they don't alter the basic rules of the language. Item 5 explains that two of the four member functions that compilers may generate are the copy constructor and the copy assignment operator. `tr1::shared_ptr` declares a generalized copy constructor, and it's clear that when the types `T` and `Y` are the same, the generalized copy constructor could be instantiated to create the “normal” copy constructor. So will compilers generate a copy constructor for `tr1::shared_ptr`, or will they instantiate the generalized copy constructor template when one `tr1::shared_ptr` object is constructed from another `tr1::shared_ptr` object of the same type?

As I said, member templates don't change the rules of the language, and the rules state that if a copy constructor is needed and you don't declare one, one will be generated for you automatically. Declaring a generalized copy constructor (a member template) in a class doesn't keep compilers from generating their own copy constructor (a non-template), so if you want to control all aspects of copy construction, you must declare both a generalized copy constructor as well as the "normal" copy constructor. The same applies to assignment. Here's an excerpt from `tr1::shared_ptr`'s definition that exemplifies this:

Click here to view code image

```
template<class T> class shared_ptr {
public:
  shared_ptr(shared_ptr const& r);                 // copy
constructor

  template<class Y>                                //
generalized
    shared_ptr(shared_ptr<Y> const& r);            // copy
constructor

  shared_ptr& operator=(shared_ptr const& r);      // copy
assignment

  template<class Y>                                //
generalized
    shared_ptr& operator=(shared_ptr<Y> const& r); // copy
assignment
  ...
};
```

Things to Remember

- Use member function templates to generate functions that accept all compatible types.
- If you declare member templates for generalized copy construction or generalized assignment, you'll still need to

declare the normal copy constructor and copy assignment operator, too.

Item 46: Define non-member functions inside templates when type conversions are desired

Item 24 explains why only non-member functions are eligible for implicit type conversions on all arguments, and it uses as an example the `operator*` function for a `Rational` class. I recommend you familiarize yourself with that example before continuing, because this Item extends the discussion with a seemingly innocuous modification to Item 24's example: it templatizes both `Rational` and `operator*`:

Click here to view code image

```
template<typename T>
class Rational {
public:
  Rational(const T& numerator = 0,     // see Item 20 for why
params
           const T& denominator = 1);  // are now passed by
reference

  const T numerator() const;           // see Item 28 for why
return
  const T denominator() const;         // values are still
passed by value,
  ...                                  // Item 3 for why
they're const
};

template<typename T>
const Rational<T> operator*(const Rational<T>& lhs,
                            const Rational<T>& rhs)
{ ... }
```

As in Item 24, we want to support mixed-mode arithmetic, so we want the code below to compile. We expect that it will, because we're using the same code that works in Item 24. The only difference is that `Rational` and operator`*` are now templates:

Click here to view code image

```
Rational<int> oneHalf(1, 2);          // this example is from
Item 24,
                                      // except Rational is
now a template

Rational<int> result = oneHalf * 2;   // error! won't compile
```

The fact that this fails to compile suggests that there's something about the templatized `Rational` that's different from the non-template version, and indeed there is. In Item 24, compilers know what function we're trying to call (`operator*` taking two `Rational`s), but here, compilers do *not* know which function we want to call. Instead, they're trying to *figure out* what function to instantiate (i.e., create) from the template named `operator*`. They know that they're supposed to instantiate some function named `operator*` taking two parameters of type `Rational<T>`, but in order to do the instantiation, they have to figure out what `T` is. The problem is, they can't.

In attempting to deduce `T`, they look at the types of the arguments being passed in the call to `operator*`. In this case, those types are `Rational<int>` (the type of `oneHalf`) and `int` (the type of 2). Each parameter is considered separately.

The deduction using `oneHalf` is easy. `operator*`'s first parameter is declared to be of type `Rational<T>`, and the first argument passed to `operator*` (`oneHalf`) is of type `Rational<int>`, so `T` must be `int`. Unfortunately, the deduction for the other parameter is not so simple. `operator*`'s second parameter is declared to be of type `Rational<T>`, but the second argument passed to `operator*` (2) is of type `int`. How are compilers to figure out what `T` is in this case? You might expect them to use `Rational<int>`'s non-`explicit` constructor to convert 2 into a `Rational<int>`, thus allowing them to deduce that `T` is `int`, but they don't do that. They don't, because implicit type conversion functions are *never* considered during template argument deduction. Never. Such conversions are used during function calls, yes, but before you can call a function, you have to know which functions exist. In order to know that, you have to deduce parameter types for the relevant function templates (so that you can instantiate the appropriate functions). But implicit type conversion via constructor calls is not considered during template argument deduction. Item 24 involves no templates, so template argument deduction is not an issue. Now that we're in the template part of C++ (see Item 1), it's the primary issue.

We can relieve compilers of the challenge of template argument deduction by taking advantage of the fact that a `friend` declaration in a template class can refer to a specific function. That means the class `Rational<T>` can declare `operator*` for `Rational<T>` as a friend function. Class templates don't depend on template argument deduction (that process applies only to function templates), so `T` is always known at the time the class `Rational<T>` is instantiated. That makes it easy for the `Rational<T>` class to declare the appropriate `operator*` function as a friend:

Click here to view code image

```
template<typename T>
class Rational {
public:
  ...

friend                                              //
declare operator*
  const Rational operator*(const Rational& lhs,     //
function (see
                           const Rational& rhs);    // below
for details)
};

template<typename T>                                // define
operator*
const Rational<T> operator*(const Rational<T>& lhs, //
functions
                            const Rational<T>& rhs)
{ ... }
```

Now our mixed-mode calls to `operator*` will compile, because when the object `oneHalf` is declared to be of type `Rational<int>`, the class `Rational<int>` is instantiated, and as part of that process, the friend function `operator*` that takes `Rational<int>` parameters is automatically declared. As a declared *function* (not a function *template*), compilers can use implicit conversion functions (such as `Rational`'s non-`explicit` constructor) when calling it, and that's how they make the mixed-mode call succeed.

Alas, “succeed” is a funny word in this context, because although the code will compile, it won't link. We'll deal with that in a moment, but first I want to remark on the syntax used to declare `operator*` inside `Rational`.

Inside a class template, the name of the template can be used as shorthand for the template and its parameters, so inside `Rational<T>`, we can just write `Rational` instead of `Rational<T>`. That saves us only a few characters in this example, but when there are multiple parameters or longer parameter names, it can both save typing and make the resulting code clearer. I bring this up, because `operator*` is declared taking and returning `Rational`s instead of `Rational<T>`s. It would have been just as valid to declare `operator*` like this:

Click here to view code image

```
template<typename T>
class Rational {
public:
  ...
friend
   const Rational<T> operator*(const Rational<T>& lhs,
                               const Rational<T>& rhs);
  ...
};
```

However, it's easier (and more common) to use the shorthand form.

Now back to the linking problem. The mixed-mode code compiles, because compilers know that we want to call a specific function (`operator*` taking a `Rational<int>` and a `Rational<int>`), but that function is only *declared* inside `Rational`, not *defined* there. Our intent is to have the `operator*` template outside the class provide that definition, but things don't work that way. If we declare a function ourselves (which is what we're doing inside the `Rational` template), we're also responsible for defining that function. In this case, we never provide a definition, and that's why linkers can't find one.

The simplest thing that could possibly work is to merge the body of `operator*` into its declaration:

Click here to view code image

```
template<typename T>
class Rational {
public:
  ...
friend const Rational operator*(const Rational& lhs, const
Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),       //
same impl
                  lhs.denominator() * rhs.denominator());  //
as in
}                                                          //
Item 24
};
```

Indeed, this works as intended: mixed-mode calls to `operator*` now compile, link, and run. Hooray!

An interesting observation about this technique is that the use of friendship has nothing to do with a need to access non-public parts of the class. In order to make type conversions possible on all arguments, we need a non-member function (Item 24 still applies); and in order to have the proper function automatically instantiated, we need to declare the function inside the class. The only way to declare a non-member function inside a class is to make it a friend. So that's what we do. Unconventional? Yes. Effective? Without a doubt.

As Item 30 explains, functions defined inside a class are implicitly declared `inline`, and that includes friend functions like `operator*`. You can minimize the impact of such `inline` declarations by having `operator*` do nothing but call a helper function defined outside of the class. In the example in this Item, there's not much point in doing that, because `operator*` is already implemented as a one-line function, but for more complex function bodies, it may be desirable. It's worth taking a look at the “have the friend call a helper” approach.

The fact that `Rational` is a template means that the helper function will usually also be a template, so the code in the header file defining `Rational` will typically look something like this:

Click here to view code image

```
template<typename T> class Rational;                 //
declare
                                                     //
Rational
```

```
                                                     //
template
template<typename T>                                    //
declare
const Rational<T> doMultiply(const Rational<T>& lhs,    //
helper
                             const Rational<T>& rhs);   //
template
template<typename T>
class Rational {
public:
  ...

friend
  const Rational<T> operator*(const Rational<T>& lhs,
                              const Rational<T>& rhs)   //
Have friend
  { return doMultiply(lhs, rhs); }                      //
call helper
  ...
};
```

Many compilers essentially force you to put all template definitions in header files, so you may need to define `doMultiply` in your header as well. (As Item 30 explains, such templates need not be inline.) That could look like this:

Click here to view code image

```
template<typename T>                                      //
define
const Rational<T> doMultiply(const Rational<T>& lhs,      //
helper
                             const Rational<T>& rhs)      //
template in
{                                                         //
header file,
  return Rational<T>(lhs.numerator() * rhs.numerator(),   //
if necessary
                     lhs.denominator() * rhs.denominator());
}
```

As a template, of course, `doMultiply` won't support mixed-mode multiplication, but it doesn't need to. It will only be called by `operator*`, and `operator*` *does* support mixed-mode operations! In essence, the *function* `operator*` supports whatever type conversions are necessary to ensure that two `Rational` objects are being multiplied, then it passes these two objects to an appropriate instantiation of the `doMultiply` *template* to do the actual multiplication. Synergy in action, no?

Things to Remember

- When writing a class template that offers functions related to the template that support implicit type conversions on all parameters, define those functions as friends inside the class template.

Item 47: Use traits classes for information about types

The STL is primarily made up of templates for containers, iterators, and algorithms, but it also has a few utility templates. One of these is called `advance`. `advance` moves a specified iterator a specified distance:

Click here to view code image

```
template<typename IterT, typename DistT>       // move iter d
units
void advance(IterT& iter, DistT d);            // forward; if
d < 0,
                                               // move iter
backward
```

Conceptually, `advance` just does `iter += d`, but `advance` can't be implemented that way, because only random access iterators support the `+=` operation. Less powerful iterator types have to implement advance by iteratively applying `++` or `--` `d` times.

Um, you don't remember your STL iterator categories? No problem, we'll do a mini-review. There are five categories of iterators, corresponding to the operations they support. *Input iterators* can move only forward, can move only one step at a time, can only read what they point to, and can read what they're pointing to only once. They're modeled on the read pointer into an input file; the C++ library's `istream_iterator`s are representative of this category. *Output iterators* are analogous, but for output: they move only forward, move only one step at a time, can only write what they point to, and can write it only once. They're modeled on the write pointer into an output file; `ostream_iterator`s epitomize this category. These are the two least powerful iterator categories. Because input and output iterators can move only forward and can read or write what they point to at most once, they are suitable only for one-pass algorithms.

A more powerful iterator category consists of *forward iterators*. Such iterators can do everything input and output iterators can do, plus they can read or write what they point to more than once. This makes them viable for multi-pass algorithms. The STL offers no singly linked list, but some libraries offer one (usually called `slist`), and iterators into such containers are forward iterators. Iterators into TR1's hashed containers (see Item 54) may also be in the forward category.

Bidirectional iterators add to forward iterators the ability to move backward as well as forward. Iterators for the STL's `list` are in this category, as are iterators for `set`, `multiset`, `map`, and `multimap`.

The most powerful iterator category is that of *random access iterators*. These kinds of iterators add to bidirectional iterators the ability to perform “iterator arithmetic,” i.e., to jump forward or backward an arbitrary distance in constant time. Such arithmetic is analogous to pointer arithmetic, which is not surprising, because random access iterators are modeled on built-in pointers, and built-in pointers can act as random access iterators. Iterators for `vector`, `deque`, and `string` are random access iterators.

For each of the five iterator categories, C++ has a “tag struct” in the standard library that serves to identify it:

Click here to view code image

```
struct input_iterator_tag {};

struct output_iterator_tag {};

struct forward_iterator_tag: public input_iterator_tag {};

struct bidirectional_iterator_tag: public
forward_iterator_tag {};

struct random_access_iterator_tag: public
```

```
bidirectional_iterator_tag {};
```

The inheritance relationships among these structs are valid is-a relationships (see Item 32): it's true that all forward iterators are also input iterators, etc. We'll see the utility of this inheritance shortly.

But back to `advance`. Given the different iterator capabilities, one way to implement `advance` would be to use the lowest-common-denominator strategy of a loop that iteratively increments or decrements the iterator. However, that approach would take linear time. Random access iterators support constant-time iterator arithmetic, and we'd like to take advantage of that ability when it's present.

What we really want to do is implement `advance` essentially like this:

Click here to view code image

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (iter is a random access iterator) {
     iter += d;                                      // use
iterator arithmetic
  }                                                  // for
random access iters
  else {
    if (d >= 0) { while (d--) ++iter; }              // use
iterative calls to
    else { while (d++) --iter; }                     // ++ or
-- for other
  }                                                  //
iterator categories
}
```

This requires being able to determine whether `iter` is a random access iterator, which in turn requires knowing whether its type, `IterT`, is a random access iterator type. In other words, we need to get some information about a type. That's what *traits* let you do: they allow you to get information about a type during compilation.

Traits aren't a keyword or a predefined construct in C++; they're a technique and a convention followed by C++ programmers. One of the demands made on the technique is that it has to work as well for built-in types as it does for user-defined types. For example, if `advance` is called with a pointer (like a `const char*`) and an `int`, `advance` has to work, but that means that the traits technique must apply to built-in types like pointers.

The fact that traits must work with built-in types means that things like nesting information inside types won't do, because there's no way to nest information inside pointers. The traits information for a type, then, must be external to the type. The standard technique is to put it into a template and one or more specializations of that template. For iterators, the template in the standard library is named `iterator_traits`:

[Click here to view code image]

```
template<typename IterT>          // template for information
about
struct iterator_traits;           // iterator types
```

As you can see, `iterator_traits` is a struct. By convention, traits are always implemented as structs. Another convention is that the structs used to implement traits are known as — I am not making this up — traits *classes*.

The way `iterator_traits` works is that for each type `IterT`, a typedef named `iterator_category` is declared in the struct `iterator_traits<IterT>`. This typedef identifies the iterator category of `IterT`.

`iterator_traits` implements this in two parts. First, it imposes the requirement that any user-defined iterator type must contain a nested typedef named `iterator_category` that identifies the appropriate tag struct. `deque`'s iterators are random access, for example, so a class for `deque` iterators would look something like this:

[Click here to view code image]

```
template < ... >                    // template params elided
class deque {
public:
  class iterator {
  public:
    typedef random_access_iterator_tag iterator_category;
    ...
  };
  ...
};
```

`list`'s iterators are bidirectional, however, so they'd do things this way:

Click here to view code image

```
template < ... >
class list {
public:
  class iterator {
  public:
    typedef bidirectional_iterator_tag iterator_category;
    ...
  };
  ...
};
```

`iterator_traits` just parrots back the iterator class's nested typedef:

Click here to view code image

```
// the iterator_category for type IterT is whatever IterT
says it is;
// see Item 42 for info on the use of "typedef typename"
template<typename IterT>
struct iterator_traits {
  typedef typename IterT::iterator_category
iterator_category;
  ...
};
```

This works well for user-defined types, but it doesn't work at all for iterators that are pointers, because there's no such thing as a pointer with a nested typedef. The second part of the `iterator_traits` implementation handles iterators that are pointers.

To support such iterators, `iterator_traits` offers a *partial template specialization* for pointer types. Pointers act as random access iterators, so that's the category `iterator_traits` specifies for them:

Click here to view code image

```
template<typename T>                 // partial template
specialization
struct iterator_traits<T*>           // for built-in pointer
types

{
  typedef random_access_iterator_tag iterator_category;
  ...
};
```

At this point, you know how to design and implement a traits class:

- Identify some information about types you'd like to make available (e.g., for iterators, their iterator category).
- Choose a name to identify that information (e.g., `iterator_category`).
- Provide a template and set of specializations (e.g., `iterator_traits`) that contain the information for the types you want to support.

Given `iterator_traits` — actually `std::iterator_traits`, since it's part of C++'s standard library — we can refine our pseudocode for `advance`:

Click here to view code image

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (typeid(typename
std::iterator_traits<IterT>::iterator_category) ==
     typeid(std::random_access_iterator_tag))
  ...
}
```

Although this looks promising, it's not what we want. For one thing, it will lead to compilation problems, but we'll explore that in Item 48; right now, there's a more fundamental issue to consider. `IterT`'s type is known during compilation, so `iterator_traits<IterT>::iterator_category` can also be determined during compilation. Yet the `if` statement is evaluated at runtime. (unless your optimizer is crafty enough to get rid of it). Why do something at runtime that we can do during compilation? It wastes time (literally), and it bloats our executable.

What we really want is a conditional construct (i.e., an `if...else` statement) for types that is evaluated during compilation. As it happens, C++ already has a way to get that behavior. It's called overloading.

When you overload some function `f`, you specify different parameter types for the different overloads. When you call `f`, compilers pick the best overload, based on the arguments you're passing. Compilers essentially say, “If this overload is the best match for what's being passed, call this `f`; if this other overload is the best match, call it; if this third one is best, call it,” etc. See? A compile-time conditional construct for types. To get `advance` to behave the way we want, all we have to do is create multiple versions of an overloaded function containing the “guts” of `advance`, declaring each to take a different type of `iterator_category` object. I use the name `doAdvance` for these functions:

Click here to view code image

```
template<typename IterT, typename DistT>              // use
this impl for
void doAdvance(IterT& iter, DistT d,                  //
random access
               std::random_access_iterator_tag)       //
iterators

{
  iter += d;
}

template<typename IterT, typename DistT>              // use
this impl for
void doAdvance(IterT& iter, DistT d,                  //
bidirectional
               std::bidirectional_iterator_tag)       //
iterators
{
  if (d >= 0) { while (d--) ++iter; }
  else { while (d++) --iter;         }
}

template<typename IterT, typename DistT>              // use
this impl for
void doAdvance(IterT& iter, DistT d,                  //
input iterators
               std::input_iterator_tag)
{
  if (d < 0 ) {
     throw std::out_of_range("Negative distance");    // see
below
  }
  while (d--) ++iter;
}
```

Because `forward_iterator_tag` inherits from `input_iterator_tag`, the version of `doAdvance` for `input_iterator_tag` will also handle forward iterators. That's the motivation for inheritance among the various `iterator_tag` structs. (In fact, it's part of the motivation for *all* public inheritance: to be able to write code for base class types that also works for derived class types.)

The specification for `advance` allows both positive and negative distances for random access and bidirectional iterators, but behavior is undefined if you try to move a forward or input iterator a negative distance. The implementations I checked simply assumed that `d` was non-negative, thus entering a *very* long loop counting "down" to zero if a negative distance was passed in. In the code above, I've shown an exception being thrown instead. Both implementations are valid. That's the curse of undefined behavior: you *can't predict* what will happen.

Given the various overloads for `doAdvance`, all `advance` needs to do is call them, passing an extra object of the appropriate iterator category type so that the compiler will use overloading resolution to call the proper implementation:

Click here to view code image

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  doAdvance(                                              //
call the version
    iter, d,                                              //
of doAdvance
    typename                                              //
that is
      std::iterator_traits<IterT>::iterator_category()    //
appropriate for
  );                                                      //
iter's iterator
}                                                         //
category
```

We can now summarize how to use a traits class:

- Create a set of overloaded "worker" functions or function templates (e.g., `doAdvance`) that differ in a traits parameter. Implement each function in accord with the traits information passed.
- Create a "master" function or function template (e.g., `advance`) that calls the workers, passing information provided by a traits class.

Traits are widely used in the standard library. There's `iterator_traits`, of course, which, in addition to `iterator_category`, offers four other pieces of information about iterators (the most useful of which is `value_type` — Item 42 shows an example of its use). There's also `char_traits`, which holds information about character types, and `numeric_limits`, which serves up information about numeric types, e.g., their minimum and maximum representable values, etc. (The name `numeric_limits` is a bit of a surprise, because the more common convention is for traits classes to end with "traits," but `numeric_limits` is what it's called, so `numeric_limits` is the name we use.)

TR1 (see Item 54) introduces a slew of new traits classes that give information about types, including `is_fundamental<T>` (whether `T` is a built-in type), `is_array<T>` (whether `T` is an array type), and `is_base_of<T1, T2>` (whether `T1` is the same as or is a base class of `T2`). All told, TR1 adds over 50 traits classes to standard C++.

Things to Remember

- Traits classes make information about types available during compilation. They're implemented using templates and template specializations.
- In conjunction with overloading, traits classes make it possible to perform compile-time `if...else` tests on types.

Item 48: Be aware of template metaprogramming

Template metaprogramming (TMP) is the process of writing template-based C++ programs that execute during compilation. Think about that for a minute: a template metaprogram is a program written in C++ that executes *inside the C++ compiler*. When a TMP program finishes running, its output — pieces of C++ source code instantiated from templates — is then compiled as usual.

If this doesn't strike you as just plain bizarre, you're not thinking about it hard enough.

C++ was not designed for template metaprogramming, but since TMP was discovered in the early 1990s, it has proven to be so useful, extensions are likely to be added to both the language and its standard library to make TMP easier. Yes, TMP was discovered, not invented. The features underlying TMP were introduced when templates were added to C++. All that was needed was for somebody to notice how they could be used in clever and unexpected ways.

TMP has two great strengths. First, it makes some things easy that would otherwise be hard or impossible. Second, because template metaprograms execute during C++ compilation, they can shift work from runtime to compile-time. One consequence is that some kinds of errors that are usually detected at runtime can be found during compilation. Another is that C++ programs making use of TMP can be more efficient in just about every way: smaller executables, shorter runtimes, lesser memory requirements. (However, a consequence of shifting work from runtime to compile-time is that compilation takes longer. Programs using TMP may take *much* longer to compile than their non-TMP counterparts.)

Consider the pseudocode for STL's `advance` introduced on page 228. (That's in Item 47. You may want to read that Item now, because in this Item, I'll assume you are familiar with the material in that one.) As on page 228, I've highlighted the pseudo part of the code:

Click here to view code image

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (iter is a random access iterator) {
    iter += d;                                   // use
iterator arithmetic
  }                                              // for
random access iters
  else {
    if (d >= 0) { while (d--) ++iter; }          // use
iterative calls to
    else { while (d++) --iter; }                 // ++ or -
- for other
  }                                              //
iterator categories
}
```

We can use `typeid` to make the pseudocode real. That yields a “normal” C++ approach to this problem — one that does all its work at runtime:

Click here to view code image

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (typeid(typename
std::iterator_traits<IterT>::iterator_category) ==
      typeid(std::random_access_iterator_tag)) {

     iter += d;                                     // use
iterator arithmetic
  }                                                 // for
random access iters
  else {
    if (d >= 0) { while (d--) ++iter; }             // use
iterative calls to
    else { while (d++) --iter; }                    // ++ or
-- for other
  }                                                 //
iterator categories
}
```

Item 47 notes that this `typeid`-based approach is less efficient than the one using traits, because with this approach, (1) the type testing occurs at runtime instead of during compilation, and (2) the code to do the runtime type testing must be present in the executable. In fact, this example shows how TMP can be more efficient than a “normal” C++ program, because the traits approach *is* TMP. Remember, traits enable compile-time `if...else` computations on types.

I remarked earlier that some things are easier in TMP than in “normal” C++, and `advance` offers an example of that, too. Item 47 mentions that the `typeid`-based implementation of `advance` can lead to compilation problems, and here's an example where it does:

Click here to view code image

```
std::list<int>::iterator iter;

...

advance(iter, 10);                          // move iter 10
elements forward;
                                            // won't compile
with above impl.
```

Consider the version of `advance` that will be generated for the above call. After substituting `iter`'s and `10`'s types for the template parameters `IterT` and `DistT`, we get this:

Click here to view code image

```
void advance(std::list<int>::iterator& iter, int d)
{
  if
(typeid(std::iterator_traits<std::list<int>::iterator>::itera
==
      typeid(std::random_access_iterator_tag)) {

    iter += d;                                          // //
error! won’t compile
  }
  else {
    if (d >= 0) { while (d--) ++iter; }
    else { while (d++) --iter; }
  }
}
```

The problem is the highlighted line, the one using `+=`. In this case, we're trying to use `+=` on a `list<int>::iterator`, but `list<int>::iterator` is a bidirectional iterator (see Item 47), so it doesn't support `+=`. Only random access iterators support `+=`. Now, we know we'll never try to execute the `+=` line, because the `typeid` test will always fail for `list<int>::iterator`s, but compilers are obliged to make sure that all source code is valid, even if it's not executed, and “`iter += d`” isn't valid when `iter` isn't a random access iterator. Contrast this with the traits-based TMP solution, where code for different types is split into separate functions, each of which uses only operations applicable to the types for which it is written.

TMP has been shown to be Turing-complete, which means that it is powerful enough to compute anything. Using TMP, you can declare variables, perform loops, write and call functions, etc. But such constructs look very different from their “normal” C++ counterparts. For example, Item 47 shows how `if...else` conditionals in TMP are expressed via templates and template specializations. But that's assembly-level TMP. Libraries for TMP (e.g., Boost's MPL — see Item 55) offer a higher-level syntax, though still not something you'd mistake for “normal” C++.

For another glimpse into how things work in TMP, let's look at loops. TMP has no real looping construct, so the effect of loops is accomplished via recursion. (If you're not comfortable with recursion, you'll need to address that before venturing into TMP. It's largely a functional language, and recursion is to functional languages as TV is to American pop culture: inseparable.) Even the recursion isn't the normal kind, however, because TMP loops don't involve recursive function calls, they involve recursive *template instantiations*.

The “hello world” program of TMP is computing a factorial during compilation. It's not a very exciting program, but then again, neither is “hello world,” yet both are helpful as language introductions. TMP factorial computation demonstrates looping through recursive template instantiation. It also demonstrates one way in which variables are created and used in TMP. Look:

Click here to view code image

```
template<unsigned n>                 // general case: the
value of
struct Factorial {                   // Factorial<n> is n
times the value
                                     // of Factorial<n-1>

  enum { value = n * Factorial<n-1>::value };

};

template<>                           // special case: the
value of
struct Factorial<0> {                // Factorial<0> is 1
  enum { value = 1 };

};
```

Given this template metaprogram (really just the single template metafunction `Factorial`), you get the value of factorial(n) by referring to `Factorial<n>::value`.

The looping part of the code occurs where the template instantiation `Factorial<n>` references the template instantiation `Factorial<n-1>`. Like all good recursion, there's a special case that causes the recursion to terminate. Here, it's the template specialization `Factorial<0>`.

Each instantiation of the `Factorial` template is a struct, and each struct uses the enum hack (see Item 2) to declare a TMP variable named `value`. `value` is what holds the current value of the factorial computation. If TMP had a real looping construct, `value` would be updated each time around the loop. Since TMP uses recursive template instantiation in place of loops, each instantiation gets its own copy of `value`, and each copy has the proper value for its place in the "loop."

You could use `Factorial` like this:

Click here to view code image

```
int main()
{
  std::cout << Factorial<5>::value;              // prints 120
  std::cout << Factorial<10>::value;             // prints
3628800
}
```

If you think this is cooler than ice cream, you've got the makings of a template metaprogrammer. If the templates and specializations and recursive instantiations and enum hacks and the need to type things like `Factorial<n-1>::value` make your skin crawl, well, you're a pretty normal C++ programmer.

Of course, `Factorial` demonstrates the utility of TMP about as well as "hello world" demonstrates the utility of any conventional programming language. To grasp why TMP is worth knowing about, it's important to have a better understanding of what it can accomplish. Here are three examples:

- **Ensuring dimensional unit correctness.** In scientific and engineering applications, it's essential that dimensional units (e.g., mass, distance, time, etc.) be combined correctly. Assigning a variable representing mass to a variable representing velocity, for example, is an error, but dividing a distance variable by a time variable and assigning the result to a velocity variable is fine. Using TMP, it's possible to ensure (during compilation) that all dimensional unit combinations in a program are correct, no matter how complex the calculations. (This is an example of how TMP can be used for early error detection.) One interesting aspect of this use of TMP is that fractional dimensional exponents can be supported. This requires that such fractions be reduced *during compilation* so that compilers can confirm, for example, that the unit *time*$^{1/2}$ is the same as *time*$^{4/8}$.

- **Optimizing matrix operations.** Item 21 explains that some functions, including `operator*`, must return new objects, and Item 44 introduces the `SquareMatrix` class, so consider the following code:

Click here to view code image

```
typedef SquareMatrix<double, 10000> BigMatrix;
BigMatrix m1, m2, m3, m4, m5;               // create
matrices and
...                                         // give them
values

BigMatrix result = m1 * m2 * m3 * m4 * m5;    // compute
their product
```

Calculating `result` in the “normal” way calls for the creation of four temporary matrices, one for the result of each call to `operator*`. Furthermore, the independent multiplications generate a sequence of four loops over the matrix elements. Using an advanced template technology related to TMP called *expression templates*, it's possible to eliminate the temporaries and merge the loops, all without changing the syntax of the client code above. The resulting software uses less memory and runs dramatically faster.

- **Generating custom design pattern implementations.** Design patterns like Strategy (see Item 35), Observer, Visitor, etc. can be implemented in many ways. Using a TMP-based technology called *policy-based design*, it's possible to create templates representing independent design choices (“policies”) that can be combined in arbitrary ways to yield pattern implementations with custom behavior. For example, this technique has been used to allow a few templates implementing smart pointer behavioral policies to generate (during compilation) any of *hundreds* of different smart pointer types. Generalized beyond the domain of programming artifacts like design patterns and smart pointers, this technology is a basis for what's known as *generative programming*.

TMP is not for everybody. The syntax is unintuitive, and tool support is weak. (Debuggers for template metaprograms? Ha!) Being an “accidental” language that was only relatively recently discovered, TMP programming conventions are still somewhat experimental. Nevertheless, the efficiency improvements afforded by shifting work from runtime to compile-time can be impressive, and the ability to express behavior that is difficult or impossible to implement at runtime is attractive, too.

TMP support is on the rise. It's likely that the next version of C++ will provide explicit support for it, and TR1 already does (see Item 54). Books are beginning to come out on the subject, and TMP information on the web just keeps getting richer. TMP will probably never be mainstream, but for some programmers — especially library developers — it's almost certain to be a staple.

Things to Remember

- Template metaprogramming can shift work from runtime to compile-time, thus enabling earlier error detection and higher runtime performance.
- TMP can be used to generate custom code based on combinations of policy choices, and it can also be used to avoid generating code inappropriate for particular types.

8. Customizing new and delete

In these days of computing environments boasting built-in support for garbage collection (e.g., Java and .NET), the manual C++ approach to memory management can look rather old-fashioned. Yet many developers working on demanding systems applications choose C++ *because* it lets them manage memory manually. Such developers study the memory usage characteristics of their software, and they tailor their allocation and deallocation routines to offer the best possible performance (in both time and space) for the systems they build.

Doing that requires an understanding of how C++'s memory management routines behave, and that's the focus of this chapter. The two primary players in the game are the allocation and deallocation routines (`operator new` and `operator delete`), with a supporting role played by the new-handler — the function called when `operator new` can't satisfy a request for memory.

Memory management in a multithreaded environment poses challenges not present in a single-threaded system, because both the heap and the new-handler are modifiable global resources, subject to the race conditions that can bedevil threaded systems. Many Items in this chapter mention the use of modifiable static data, always something to put thread-aware programmers on high alert. Without proper synchronization, the use of lock-free algorithms, or careful design to prevent concurrent access, calls to memory routines can easily lead to corrupted heap management data structures. Rather than repeatedly remind you of this danger, I'll mention it here and assume that you keep it in mind for the rest of the chapter.

Something else to keep in mind is that `operator new` and `operator delete` apply only to allocations for single objects. Memory for arrays is allocated by `operator new[]` and deallocated by `operator delete[]`. (In both cases, note the “`[]`” part of the function names.) Unless indicated otherwise, everything I write about `operator new` and `operator delete` also applies to `operator new[]` and `operator delete[]`.

Finally, note that heap memory for STL containers is managed by the containers' allocator objects, not by `new` and `delete` directly. That being the case, this chapter has nothing to say about STL allocators.

Item 49: Understand the behavior of the new-handler

When `operator new` can't satisfy a memory allocation request, it throws an exception. Long ago, it returned a null pointer, and some older compilers still do that. You can still get the old behavior (sort of), but I'll defer that discussion until the end of this Item.

Before `operator new` throws an exception in response to an unsatisfiable request for memory, it calls a client-specifiable error-handling function called a *new-handler*. (This is not quite true. What `operator new` really does is a bit more complicated. Details are provided in Item 51.) To specify the out-of-memory-handling function, clients call `set_new_handler`, a standard library function declared in `<new>`:

Click here to view code image

```
namespace std {

  typedef void (*new_handler)();
  new_handler set_new_handler(new_handler p) throw();
}
```

As you can see, `new_handler` is a typedef for a pointer to a function that takes and returns nothing, and `set_new_handler` is a function that takes and returns a `new_handler`. (The "`throw()`" at the end of `set_new_handler`'s declaration is an exception specification. It essentially says that this function won't throw any exceptions, though the truth is a bit more interesting. For details, see Item 29.)

`set_new_handler`'s parameter is a pointer to the function `operator new` should call if it can't allocate the requested memory. The return value of `set_new_handler` is a pointer to the function in effect for that purpose before `set_new_handler` was called.

You use `set_new_handler` like this:

Click here to view code image

```
// function to call if operator new can't allocate enough
memory
void outOfMem()
{
  std::cerr << "Unable to satisfy request for memory\n";
  std::abort();
}

int main()
{
  std::set_new_handler(outOfMem);
  int *pBigDataArray = new int[100000000L];
  ...
}
```

If `operator new` is unable to allocate space for 100,000,000 integers, `outOfMem` will be called, and the program will abort after issuing an error message. (By the way, consider what happens if memory must be dynamically allocated during the course of writing the error message to `cerr`....)

When `operator new` is unable to fulfill a memory request, it calls the new-handler function repeatedly until it *can* find enough memory. The code giving rise to these repeated calls is shown in Item 51, but this high-level description is enough to conclude that a well-designed new-handler function must do one of the following:

- **Make more memory available.** This may allow the next memory allocation attempt inside `operator new` to succeed. One way to implement this strategy is to allocate a large block of memory at program start-up, then release it for use in the program the first time the new-handler is invoked.
- **Install a different new-handler.** If the current new-handler can't make any more memory available, perhaps it knows of a different new-handler that can. If so, the current new-handler can install the other new-handler in its place (by calling `set_new_handler`). The next time `operator new` calls the new-handler function, it will get the one most recently installed. (A variation on this theme is for a new-handler to modify its *own* behavior, so the next time it's invoked, it does something different. One way to achieve this is to have the new-handler modify static, namespace-specific, or global data that affects the new-handler's behavior.)
- **Deinstall the new-handler**, i.e., pass the null pointer to `set_new_handler`. With no new-handler installed, `operator new` will throw an exception when memory allocation is unsuccessful.
- **Throw an exception** of type `bad_alloc` or some type derived from `bad_alloc`. Such exceptions will not be caught by `operator new`, so they will

propagate to the site originating the request for memory.

- **Not return**, typically by calling `abort` or `exit`.

These choices give you considerable flexibility in implementing new-handler functions.

Sometimes you'd like to handle memory allocation failures in different ways, depending on the class of the object being allocated:

Click here to view code image

```
class X {
public:
  static void outOfMemory();
  ...
};
class Y {
public:
  static void outOfMemory();
  ...
};
X* p1 = new X;                          // if allocation is
unsuccessful,
                                        // call X::outOfMemory

Y* p2 = new Y;                          // if allocation is
unsuccessful,
                                        // call Y::outOfMemory
```

C++ has no support for class-specific new-handlers, but it doesn't need any. You can implement this behavior yourself. You just have each class provide its own versions of `set_new_handler` and `operator new`. The class's `set_new_handler` allows clients to specify the new-handler for the class (exactly like the standard `set_new_handler` allows clients to specify the global new-handler). The class's `operator new` ensures that the class-specific new-handler is used in place of the global new-handler when memory for class objects is allocated.

Suppose you want to handle memory allocation failures for the `Widget` class. You'll have to keep track of the function to call when `operator new` can't allocate enough memory for a `Widget` object, so you'll declare a static member of type `new_handler` to point to the new-handler function for the class. `Widget` will look something like this:

Click here to view code image

```
class Widget {
public:
  static std::new_handler set_new_handler(std::new_handler p)
throw();
  static void* operator new(std::size_t size)
throw(std::bad_alloc);
private:
  static std::new_handler currentHandler;
};
```

Static class members must be defined outside the class definition (unless they're `const` and integral—see Item 2), so:

Click here to view code image

```
std::new_handler Widget::currentHandler = 0;    // init to
null in the class
                                                // impl. file
```

The `set_new_handler` function in `Widget` will save whatever pointer is passed to it, and it will return whatever pointer had been saved prior to the call. This is what the standard version of `set_new_handler` does:

Click here to view code image

```
std::new_handler Widget::set_new_handler(std::new_handler p)
throw()
{
  std::new_handler oldHandler = currentHandler;
  currentHandler = p;
  return oldHandler;
}
```

Finally, `Widget`'s `operator new` will do the following:

1. Call the standard `set_new_handler` with `Widget`'s error-handling function. This installs `Widget`'s new-handler as the global new-handler.
2. Call the global `operator new` to perform the actual memory allocation. If allocation fails, the global `operator new` invokes `Widget`'s new-handler, because that function was just installed as the global new-handler. If the global `operator new` is ultimately unable to allocate the memory, it throws

a `bad_alloc` exception. In that case, `Widget`'s `operator new` must restore the original global new-handler, then propagate the exception. To ensure that the original new-handler is always reinstated, `Widget` treats the global new-handler as a resource and follows the advice of Item 13 to use resource-managing objects to prevent resource leaks.

3. If the global `operator new` was able to allocate enough memory for a `Widget` object, `Widget`'s `operator new` returns a pointer to the allocated memory. The destructor for the object managing the global new-handler automatically restores the global new-handler to what it was prior to the call to `Widget`'s `operator new`.

Here's how you say all that in C++. We'll begin with the resource-handling class, which consists of nothing more than the fundamental RAII operations of acquiring a resource during construction and releasing it during destruction (see Item 13):

Click here to view code image

```
class NewHandlerHolder {
public:
  explicit NewHandlerHolder(std::new_handler nh)    //
acquire current
  : handler(nh) {}                                   // new-
handler

  ~NewHandlerHolder()                               //
release it
  { std::set_new_handler(handler); }
private:
  std::new_handler handler;                         //
remember it

  NewHandlerHolder(const NewHandlerHolder&);        //
prevent copying
  NewHandlerHolder&                                 // (see
Item 14)
   operator=(const NewHandlerHolder&);
};
```

This makes implementation of `Widget`'s `operator new` quite simple:

Click here to view code image

```
void* Widget::operator new(std::size_t size)
throw(std::bad_alloc)
{
  NewHandlerHolder                              // install
Widget's
   h(std::set_new_handler(currentHandler));     // new-
handler

  return ::operator new(size);                  // allocate
memory
                                                // or throw

}                                               // restore
global
                                                // new-
handler
```

Clients of `Widget` use its new-handling capabilities like this:

Click here to view code image

```
void outOfMem();                   // decl. of func. to call
if mem. alloc.
                                   // for Widget objects
fails

Widget::set_new_handler(outOfMem); // set outOfMem as
Widget's
                                   // new-handling function

Widget *pw1 = new Widget;          // if memory allocation
                                   // fails, call outOfMem

std::string *ps = new std::string; // if memory allocation
fails,
                                   // call the global new-
handling
                                   // function (if there is
one)

Widget::set_new_handler(0);        // set the Widget-specific
                                   // new-handling function
to
                                   // nothing (i.e., null)
```

```
Widget *pw2 = new Widget;          // if mem. alloc. fails,
throw an
                                   // exception immediately.
(There is
                                   // no new- handling
function for
                                   // class Widget.)
```

The code for implementing this scheme is the same regardless of the class, so a reasonable goal would be to reuse it in other places. An easy way to make that possible is to create a “mixin-style” base class, i.e., a base class that's designed to allow derived classes to inherit a single specific capability — in this case, the ability to set a class-specific new-handler. Then turn the base class into a template, so that you get a different copy of the class data for each inheriting class.

The base class part of this design lets derived classes inherit the `set_new_handler` and `operator new` functions they all need, while the template part of the design ensures that each inheriting class gets a different `currentHandler` data member. That may sound a bit complicated, but the code looks reassuringly familiar. In fact, the only real difference is that it's now available to any class that wants it:

Click here to view code image

```
template<typename T>              // "mixin-style" base class
for
class NewHandlerSupport{          {// class-specific
set_new_handler
public:                           // support

  static std::new_handler set_new_handler(std::new_handler p)
throw();
  static void* operator new(std::size_t size)
throw(std::bad_alloc);

  ...                             // other versions of op.
new —
                                  // see Item 52
private:
  static std::new_handler currentHandler;
};

template<typename T>
std::new_handler
NewHandlerSupport<T>::set_new_handler(std::new_handler p)
throw()
{
std::new_handler oldHandler = currentHandler;
currentHandler = p;
return oldHandler;
}

template<typename T>
void* NewHandlerSupport<T>::operator new(std::size_t size)
  throw(std::bad_alloc)
{
  NewHandlerHolder h(std::set_new_handler(currentHandler));
  return ::operator new(size);
}
// this initializes each currentHandler to null
template<typename T>
std::new_handler NewHandlerSupport<T>::currentHandler = 0;
```

With this class template, adding `set_new_handler` support to `Widget` is easy: `Widget` just inherits from `NewHandlerSupport<Widget>`. (That may look peculiar, but I'll explain in more detail below exactly what's going on.)

Click here to view code image

```
class Widget: public NewHandlerSupport<Widget> {
  ...                              // as before, but without
declarations for
};                                 // set_new_handler or operator
new
```

That's all `Widget` needs to do to offer a class-specific `set_new_handler`.

But maybe you're still fretting over `Widget` inheriting from `NewHandlerSupport<Widget>`. If so, your fretting may intensify when you note that the `NewHandlerSupport` template never uses its type parameter `T`. It doesn't need to. All we need is a different copy of `NewHandlerSupport` — in particular, its static data member `currentHandler` — for each class that inherits from `NewHandlerSupport`. The template parameter `T` just distinguishes one inheriting class from another. The template mechanism itself automatically generates a copy of `currentHandler` for each `T` with which `NewHandlerSupport` is instantiated.

As for `Widget` inheriting from a templatized base class that takes `Widget` as a type parameter, don't feel bad if the notion makes you a little woozy. It initially has that effect on everybody. However, it turns out to be such a useful technique, it has a name, albeit one that reflects the fact that it looks natural to no one the first time they see it. It's called the *curiously recurring template pattern* (CRTP). Honest.

At one point, I published an article suggesting that a better name would be “Do It For Me,” because when `Widget` inherits from `NewHandlerSupport<Widget>`, it's really saying, “I'm `Widget`, and I want to inherit from the `NewHandlerSupport` class for `Widget`.” Nobody uses my proposed name (not even me), but thinking about CRTP as a way of saying “do it for me” may help you understand what the templatized inheritance is doing.

Templates like `NewHandlerSupport` make it easy to add a class-specific new-handler to any class that wants one. Mixin-style inheritance, however, invariably leads to the topic of multiple inheritance, and before starting down that path, you'll want to read Item 40.

Until 1993, C++ required that `operator new` return null when it was unable to allocate the requested memory. `operator new` is now specified to throw a `bad_alloc` exception, but a lot of C++ was written before compilers began supporting the revised specification. The C++ standardization committee didn't want to abandon the test-for-null code base, so they provided alternative forms of `operator new` that offer the traditional failure-yields-null behavior. These forms are called “nothrow” forms, in part because they employ `nothrow` objects (defined in the header `<new>`) at the point where `new` is used:

Click here to view code image

```
class Widget { ... };
Widget *pw1 = new Widget;                 // throws bad_alloc
if
                                          // allocation fails

if (pw1 == 0) ...                         // this test must
fail

Widget *pw2 =new (std::nothrow) Widget;   // returns 0 if
```

```
allocation for
                                          // the Widget fails

if (pw2 == 0) ...                         // this test may
succeed
```

Nothrow `new` offers a less compelling guarantee about exceptions than is initially apparent. In the expression “`new (std::nothrow) Widget`,” two things happen. First, the nothrow version of `operator new` is called to allocate enough memory for a `Widget` object. If that allocation fails, `operator new` returns the null pointer, just as advertised. If it succeeds, however, the `Widget` constructor is called, and at that point, all bets are off. The `Widget` constructor can do whatever it likes. It might itself `new` up some memory, and if it does, it's not constrained to use nothrow `new`. Although the `operator new` call in “`new (std::nothrow) Widget`” won't throw, then, the `Widget` constructor might. If it does, the exception will be propagated as usual. Conclusion? Using nothrow `new` guarantees only that `operator new` won't throw, not that an expression like “`new (std::nothrow) Widget`” will never yield an exception. In all likelihood, you will never have a need for nothrow `new`.

Regardless of whether you use “normal” (i.e., exception-throwing) `new` or its somewhat stunted nothrow cousin, it's important that you understand the behavior of the new-handler, because it's used with both forms.

Things to Remember

- `set_new_handler` allows you to specify a function to be called when memory allocation requests cannot be satisfied.
- Nothrow `new` is of limited utility, because it applies only to memory allocation; associated constructor calls may still throw exceptions.

Item 50: Understand when it makes sense to replace `new` and `delete`

Let's return to fundamentals for a moment. Why would anybody want to replace the compiler-provided versions of `operator new` or `operator delete` in the first place? These are three of the most common reasons:

- **To detect usage errors.** Failure to `delete` memory conjured up by `new` leads to memory leaks. Using more than one `delete` on `new`ed memory yields undefined behavior. If `operator new` keeps a list of allocated addresses and `operator delete` removes addresses from the list, it's easy to detect such usage errors. Similarly, a variety of programming mistakes can lead to data overruns (writing beyond the end of an allocated block) and underruns (writing prior to the beginning of an allocated block). Custom `operator new`s can overallocate blocks so there's room to put known byte patterns ("signatures") before and after the memory made available to clients. `operator delete`s can check to see if the signatures are still intact. If they're not, an overrun or underrun occurred sometime during the life of the allocated block, and `operator delete` can log that fact, along with the value of the offending pointer.

- **To improve efficiency.** The versions of `operator new` and `operator delete` that ship with compilers are designed for general-purpose use. They have to be acceptable for long-running programs (e.g., web servers), but they also have to be acceptable for programs that execute for less than a second. They have to handle series of requests for large blocks of memory, small blocks, and mixtures of the two. They have to accommodate allocation patterns ranging from the dynamic allocation of a few blocks that exist for the duration of the program to constant allocation and deallocation of a large number of short-lived objects. They have to worry about heap fragmentation, a process that, if unchecked, eventually leads to the inability to satisfy requests for large blocks of memory, even when ample free memory is distributed across many small blocks.

  Given the demands made on memory managers, it's no surprise that the `operator new`s and `operator delete`s that ship with compilers take a middle-of-the-road strategy. They work reasonably well for everybody, but optimally for nobody. If you have a good understanding of your program's dynamic memory usage patterns, you can often find that custom versions of `operator new` and `operator delete` outperform the default ones. By "outperform," I mean they run faster — sometimes orders of magnitude faster — and they require less memory — up to 50% less. For some (though by no means all) applications, replacing the stock `new` and `delete` with custom versions is an easy way to pick up significant performance improvements.

- **To collect usage statistics.** Before heading down the path of writing custom `new`s and `delete`s, it's prudent to gather information about how your software uses its dynamic memory. What is the distribution of allocated block sizes? What is the distribution of their lifetimes? Do they tend to be allocated and deallocated in FIFO (“first in, first out”) order, LIFO (“last in, first out”) order, or something closer to random order? Do the usage patterns change over time, e.g., does your software have different allocation/deallocation patterns in different stages of execution? What is the maximum amount of dynamically allocated memory in use at any one time (i.e., its “high water mark”)? Custom versions of `operator new` and `operator delete` make it easy to collect this kind of information.

In concept, writing a custom `operator new` is pretty easy. For example, here's a quick first pass at a global `operator new` that facilitates the detection of under- and overruns. There are a lot of little things wrong with it, but we'll worry about those in a moment.

Click here to view code image

```
static const int signature = 0xDEADBEEF;

typedef unsigned char Byte;

// this code has several flaws—see below
void* operator new(std::size_t size) throw(std::bad_alloc)
{
  using namespace std;

  size_t realSize = size + 2 * sizeof(int);    // increase
size of request so 2
                                               // signatures
will also fit inside

  void *pMem = malloc(realSize);               // call malloc
to get the actual
  if (!pMem) throw bad_alloc();                // memory

  // write signature into first and last parts of the memory
  *(static_cast<int*>(pMem)) = signature;
  *(reinterpret_cast<int*>(static_cast<Byte*>(pMem)+realSize-
sizeof(int))) =
  signature;

  // return a pointer to the memory just past the first
signature
  return static_cast<Byte*>(pMem) + sizeof(int);
}
```

Most of the shortcomings of this `operator new` have to do with its failure to adhere to the C++ conventions for functions of that name. For example, Item 51 explains that all `operator new`s should contain a loop calling a new-handling function, but this one doesn't. However, Item 51 is devoted to such conventions, so I'll ignore them here. I want to focus on a more subtle issue now: *alignment*.

Many computer architectures require that data of particular types be placed in memory at particular kinds of addresses. For example, an architecture might require that pointers occur at addresses that are a multiple of four (i.e., be *four-byte aligned*) or that `double`s must occur at addresses that are a multiple of eight (i.e., be *eight-byte aligned*). Failure to follow such constraints could lead to hardware exceptions at runtime. Other architectures are more forgiving, though they may offer better performance if alignment preferences are satisfied. For example, `double`s may be aligned on any byte boundary on the Intel x86 architecture, but access to them is a lot faster if they are eight-byte aligned.

Alignment is relevant here, because C++ requires that all `operator new`s return pointers that are suitably aligned for *any* data type. `malloc` labors under the same requirement, so having `operator new` return a pointer it gets from `malloc` is safe. However, in `operator new` above, we're not returning a pointer we got from `malloc`, we're returning a pointer we got from `malloc` *offset by the size of an* `int`. There is no guarantee that this is safe! If the client called `operator new` to get enough memory for a `double` (or, if we were writing `operator new[]`, an array of `double`s) and we were running on a machine where `int`s were four bytes in size but `double`s were required to be eight-byte aligned, we'd probably return a pointer with improper alignment. That might cause the program to crash. Or it might just cause it to run more slowly. Either way, it's probably not what we had in mind.

Details like alignment are the kinds of things that distinguish professional-quality memory managers from ones thrown together by programmers distracted by the need to get on to other tasks. Writing a custom memory manager that almost works is pretty easy. Writing one that works *well* is a lot harder. As a general rule, I suggest you not attempt it unless you have to.

In many cases, you don't have to. Some compilers have switches that enable debugging and logging functionality in their memory management functions. A quick glance through your compilers' documentation may eliminate your need to consider writing `new` and `delete`. On many platforms, commercial products can replace the memory management functions that ship with compilers. To avail yourself of their enhanced functionality and (presumably) improved performance, all you need do is relink. (Well, you also have to buy them.)

Another option is open source memory managers. They're available for many platforms, so you can download and try those. One such open source allocator is the Pool library from Boost (see Item 55). The Pool library offers allocators tuned for one of the most common situations in which custom memory management is helpful: allocation of a large number of small objects. Many C++ books, including earlier editions of this one, show the code for a high-performance small-object allocator, but they usually omit such pesky details as portability and alignment considerations, thread safety, etc. Real libraries tend to have code that's a lot more robust. Even if you decide to write your own `new`s and `delete`s, looking at open source versions is likely to give you insights into the easy-to-overlook details that separate almost working from really working. (Given that alignment is one such detail, it's worth noting that TR1 (see Item 54) includes support for discovering type-specific alignment requirements.)

The topic of this Item is knowing when it can make sense to replace the default versions of `new` and `delete`, either globally or on a per-class basis. We're now in a position to summarize when in more detail than we did before.

- **To detect usage errors** (as above).
- **To collect statistics about the use of dynamically allocated memory** (also as above).
- **To increase the speed of allocation and deallocation.** General-purpose allocators are often (though not always) a lot slower than custom versions, especially if the custom versions are designed for objects of a particular type. Class-specific allocators are an example application of fixed-size allocators such as those offered by Boost's Pool library. If your application is single-threaded, but your compilers' default memory management routines are thread-safe, you may be able to win measurable

speed improvements by writing thread-unsafe allocators. Of course, before jumping to the conclusion that `operator new` and `operator delete` are worth speeding up, be sure to profile your program to confirm that these functions are truly a bottleneck.

- **To reduce the space overhead of default memory management.** General-purpose memory managers are often (though not always) not just slower than custom versions, they often use more memory, too. That's because they often incur some overhead for each allocated block. Allocators tuned for small objects (such as those in Boost's Pool library) essentially eliminate such overhead.
- **To compensate for suboptimal alignment in the default allocator.** As I mentioned earlier, it's fastest to access `double`s on the x86 architecture when they are eight-byte aligned. Alas, the `operator new`s that ship with some compilers don't guarantee eight-byte alignment for dynamic allocations of `double`s. In such cases, replacing the default `operator new` with one that guarantees eight-byte alignment could yield big increases in program performance.
- **To cluster related objects near one another.** If you know that particular data structures are generally used together and you'd like to minimize the frequency of page faults when working on the data, it can make sense to create a separate heap for the data structures so they are clustered together on as few pages as possible. Placement versions of `new` and `delete` (see Item 52) can make it possible to achieve such clustering.
- **To obtain unconventional behavior.** Sometimes you want `operator`s `new` and `delete` to do something that the compiler-provided versions don't offer. For example, you might want to allocate and deallocate blocks in shared memory, but have only a C API through which to manage that memory. Writing custom versions of `new` and `delete` (probably placement versions — again, see Item 52) would allow you to drape the C API in C++ clothing. As another example, you might write a custom `operator delete` that overwrites deallocated memory with zeros in order to increase the security of application data.

Things to Remember

- There are many valid reasons for writing custom versions of `new` and `delete`, including improving performance, debugging heap usage errors, and collecting heap usage information.

Item 51: Adhere to convention when writing `new` and `delete`

Item 50 explains when you might want to write your own versions of `operator new` and `operator delete`, but it doesn't explain the conventions you must follow when you do it. The rules aren't hard to follow, but some of them are unintuitive, so it's important to know what they are.

We'll begin with `operator new`. Implementing a conformant `operator new` requires having the right return value, calling the new-handling function when insufficient memory is available (see Item 49), and being prepared to cope with requests for no memory. You'll also want to avoid inadvertently hiding the “normal” form of `new`, though that's more a class interface issue than an implementation requirement; it's addressed in Item 52.

The return value part of `operator new` is easy. If you can supply the requested memory, you return a pointer to it. If you can't, you follow the rule described in Item 49 and throw an exception of type `bad_alloc`.

It's not quite that simple, however, because `operator new` actually tries to allocate memory more than once, calling the new-handling function after each failure. The assumption here is that the new-handling function might be able to do something to free up some memory. Only when the pointer to the new-handling function is null does `operator new` throw an exception.

Curiously, C++ requires that `operator new` return a legitimate pointer even when zero bytes are requested. (Requiring this odd-sounding behavior simplifies things elsewhere in the language.) That being the case, pseudocode for a non-member `operator new` looks like this:

Click here to view code image

```
void * operator new(std::size_t size) throw(std::bad_alloc)
{                                              // your operator new
might
  using namespace std;                         // take additional
params

  if (size == 0) {                             // handle 0-byte
requests
    size = 1;                                  // by treating them as
  }                                            // 1-byte requests
  while (true) {
   attempt to allocate size bytes;

    if (the allocation was successful)
       return (a pointer to the memory);

    // allocation was unsuccessful; find out what the
    // current new-handling function is (see below)
    new_handler globalHandler = set_new_handler(0);
    set_new_handler(globalHandler);

    if (globalHandler) (*globalHandler)();
    else throw std::bad_alloc();
  }
}
```

The trick of treating requests for zero bytes as if they were really requests for one byte looks slimy, but it's simple, it's legal, it works, and how often do you expect to be asked for zero bytes, anyway?

You may also look askance at the place in the pseudocode where the new-handling function pointer is set to null, then promptly reset to what it was originally. Unfortunately, there is no way to get at the new-handling function pointer directly, so you have to call `set_new_handler` to find out what it is. Crude, yes, but also effective, at least for single-threaded code. In a multithreaded environment, you'll probably need some kind of lock to safely manipulate the (global) data structures behind the new-handling function.

Item 49 remarks that `operator new` contains an infinite loop, and the code above shows that loop explicitly; "`while (true)`" is about as infinite as it gets. The only way out of the loop is for memory to be successfully allocated or for the new-handling function to do one of the things described in Item 49: make more memory available, install a different new-handler, deinstall the new-handler, throw an exception of or derived from `bad_alloc`, or fail to return. It should now be clear why the new-handler must do one of those things. If it doesn't, the loop inside `operator new` will never terminate.

Many people don't realize that `operator new` member functions are inherited by derived classes. That can lead to some interesting complications. In the pseudocode for `operator new` above, notice that the function tries to allocate `size` bytes (unless `size` is zero). That makes perfect sense, because that's the argument that was passed to the function. However, as Item 50 explains, one of the most common reasons for writing a custom memory manager is to optimize allocation for objects of a *specific* class, not for a class or any of its derived classes. That is, given an `operator new` for a class `X`, the behavior of that function is typically tuned for objects of size `sizeof(X)`—nothing larger and nothing smaller. Because of inheritance, however, it is possible that the `operator new` in a base class will be called to allocate memory for an object of a derived class:

Click here to view code image

```
class Base {
public:
  static void* operator new(std::size_t size)
throw(std::bad_alloc);
  ...
};

class Derived: public Base                // Derived doesn't
declare
{ ... };                                  // operator new

Derived *p = new Derived;                 // calls
Base::operator new!
```

If `Base`'s class-specific `operator new` wasn't designed to cope with this — and chances are that it wasn't — the best way for it to handle the situation is to slough off calls requesting the “wrong” amount of memory to the standard `operator new`, like this:

Click here to view code image

```
void * Base::operator new(std::size_t size)
throw(std::bad_alloc)
{
  if (size != sizeof(Base))               // if size is
"wrong,"
     return ::operator new(size);         // have standard
operator
                                          // new handle the
request

  ...                                     // otherwise handle
                                          // the request here
}
```

“Hold on!” I hear you cry, “You forgot to check for the pathological-but-nevertheless-possible case where `size` is zero!” Actually, I didn't, and please stop using hyphens when you cry out. The test is still there, it's just been incorporated into the test of `size` against `sizeof(Base)`. C++ works in some mysterious ways, and one of those ways is to decree that all freestanding objects have non-zero size (see Item 39). By definition, `sizeof(Base)` can never be zero, so if `size` is zero, the request will be forwarded to `::operator new`, and it will become that function's responsibility to treat the request in a reasonable fashion.

If you'd like to control memory allocation for arrays on a per-class basis, you need to implement `operator new`'s array-specific cousin, `operator new[]`. (This function is usually called “array new,” because it's hard to figure out how to pronounce “`operator new[]`”.) If you decide to write `operator new[]`, remember that all you're doing is allocating a chunk of raw memory — you can't do anything to the as-yet-nonexistent objects in the array. In fact, you can't even figure out how many objects will be in the array. First, you don't know how big each object is. After all, a base class's `operator new[]` might, through inheritance, be called to allocate memory for an array of derived class objects, and derived class objects are usually bigger than base class objects.

Hence, you can't assume inside `Base::operator new[]` that the size of each object going into the array is `sizeof(Base)`, and that means you can't assume that the number of objects in the array is `(`*`bytes requested`*`)/sizeof(Base)`. Second, the `size_t` parameter passed to `operator new[]` may be for more memory than will be filled with objects, because, as Item 16 explains, dynamically allocated arrays may include extra space to store the number of array elements.

So much for the conventions you need to follow when writing `operator new`. For `operator delete`, things are simpler. About all you need to remember is that C++ guarantees it's always safe to delete the null pointer, so you need to honor that guarantee. Here's pseudocode for a non-member `operator delete`:

Click here to view code image

```
void operator delete(void *rawMemory) throw()
{
  if (rawMemory == 0) return;            // do nothing if the
null
                                         // pointer is being
deleted

  deallocate the memory pointed to by rawMemory;
}
```

The member version of this function is simple, too, except you've got to be sure to check the size of what's being deleted. Assuming your class-specific `operator new` forwards requests of the “wrong” size to `::operator new`, you've got to forward “wrongly sized” deletion requests to `::operator delete`:

Click here to view code image

```
class Base {                            // same as before,
but now
public:                                 // operator delete is
declared

  static void* operator new(std::size_t size)
throw(std::bad_alloc);
  static void operator delete(void *rawMemory, std::size_t
size) throw();
  ...
};
void Base::operator delete(void *rawMemory, std::size_t size)
throw()
{
  if (rawMemory == 0) return;           // check for null
pointer

  if (size != sizeof(Base)) {           // if size is
"wrong,"
     ::operator delete(rawMemory);      // have standard
operator
     return;                            // delete handle the
request
  }
  deallocate the memory pointed to by rawMemory;
  return;
}
```

Interestingly, the `size_t` value C++ passes to `operator delete` may be incorrect if the object being deleted was derived from a base class lacking a virtual destructor. This is reason enough for making sure your base classes have virtual destructors, but Item 7 describes a second, arguably better reason. For now, simply note that if you omit virtual destructors in base classes, `operator delete` functions may not work correctly.

Things to Remember

- `operator new` should contain an infinite loop trying to allocate memory, should call the new-handler if it can't satisfy a memory request, and should handle requests for zero bytes. Class-specific versions should handle requests for larger blocks than expected.
- `operator delete` should do nothing if passed a pointer that is null. Class-specific versions should handle blocks that are larger than expected.

Item 52: Write placement `delete` if you write placement `new`

Placement `new` and placement `delete` aren't the most commonly encountered beasts in the C++ menagerie, so don't worry if you're not familiar with them. Instead, recall from Items 16 and 17 that when you write a `new` expression such as this,

```
Widget *pw = new Widget;
```

two functions are called: one to `operator new` to allocate memory, a second to `Widget`'s default constructor.

Suppose that the first call succeeds, but the second call results in an exception being thrown. In that case, the memory allocation performed in step 1 must be undone. Otherwise we'll have a memory leak. Client code can't deallocate the memory, because if the `Widget` constructor throws an exception, `pw` is never assigned. There'd be no way for clients to get at the pointer to the memory that should be deallocated. The responsibility for undoing step 1 must therefore fall on the C++ runtime system.

The runtime system is happy to call the `operator delete` that corresponds to the version of `operator new` it called in step 1, but it can do that only if it knows which `operator delete` — there may be many — is the proper one to call. This isn't an issue if you're dealing with the versions of `new` and `delete` that have the normal signatures, because the normal `operator new`,

```
void* operator new(std::size_t) throw(std::bad_alloc);
```

corresponds to the normal `operator delete`:

Click here to view code image

```
void operator delete(void *rawMemory) throw();  // normal
signature
                                                // at global
scope

void operator delete(void *rawMemory,           // typical
normal
                     std::size_t size) throw(); // signature
at class
                                                // scope
```

When you're using only the normal forms of `new` and `delete`, then, the runtime system has no trouble finding the `delete` that knows how to undo what `new` did. The which-`delete`-goes-with-this-`new` issue does arise, however, when you start declaring non-normal forms of `operator new` — forms that take additional parameters.

For example, suppose you write a class-specific `operator new` that requires specification of an `ostream` to which allocation information should be logged, and you also write a normal class-specific `operator delete`:

Click here to view code image

```
class Widget {
public:
  ...
  static void* operator new(std::size_t size,              //
non-normal
                            std::ostream& logStream)       //
form of new
    throw(std::bad_alloc);

  static void operator delete(void
*pMemory,                // normal class-
                              std::size_t size) throw();   //
specific form
                                                                    //
of delete
  ...
};
```

This design is problematic, but before we see why, we need to make a brief terminological detour.

When an `operator new` function takes extra parameters (other than the mandatory `size_t` argument), that function is known as a *placement* version of `new`. The `operator new` above is thus a placement version. A particularly useful placement `new` is the one that takes a pointer specifying where an object should be constructed. That `operator new` looks like this:

Click here to view code image

```
void* operator new(std::size_t, void *pMemory) throw();   //
"placement
                                                           //
new"
```

This version of `new` is part of C++'s standard library, and you have access to it whenever you `#include <new>`. Among other things, this `new` is used inside `vector` to create objects in the vector's unused capacity. It's also the *original* placement `new`. In fact, that's how this function is known: as *placement* `new`. Which means that the term "placement `new`" is overloaded. Most of the time when people talk about placement `new`, they're talking about this specific function, the `operator new` taking a single extra argument of type `void*`. Less commonly, they're talking about any version of `operator new` that takes extra arguments. Context generally clears up any ambiguity, but it's important to understand that the general term "placement `new`" means any version of `new` taking extra arguments, because the phrase "placement `delete`" (which we'll encounter in a moment) derives directly from it.

But let's get back to the declaration of the `Widget` class, the one whose design I said was problematic. The difficulty is that this class will give rise to subtle memory leaks. Consider this client code, which logs allocation information to `cerr` when dynamically creating a `Widget`:

Click here to view code image

```
Widget *pw = new (std::cerr) Widget; // call operator new,
passing cerr as
                                     // the ostream; this
leaks memory
                                     // if the Widget
constructor throws
```

Once again, if memory allocation succeeds and the `Widget` constructor throws an exception, the runtime system is responsible for undoing the allocation that `operator new` performed. However, the runtime system can't really understand how the called version of `operator new` works, so it can't undo the allocation itself. Instead, the runtime system looks for a version of `operator delete` that takes *the same number and types of extra arguments* as `operator new`, and, if it finds it, that's the one it calls. In this case, `operator new` takes an extra argument of type `ostream&`, so the corresponding `operator delete` would have this signature:

```
void operator delete(void*, std::ostream&) throw();
```

By analogy with placement versions of `new`, versions of `operator delete` that take extra parameters are known as *placement* *`delete`s*. In this case, `Widget` declares no placement version of `operator delete`, so the runtime system doesn't know how to undo what the call to placement `new` does. As a result, it does nothing. In this example, *no* *`operator delete`* *is called* if the `Widget` constructor throws an exception!

The rule is simple: if an `operator new` with extra parameters isn't matched by an `operator delete` with the same extra parameters, no `operator delete` will be called if a memory allocation by the `new` needs to be undone. To eliminate the memory leak in the code above, `Widget` needs to declare a placement `delete` that corresponds to the logging placement `new`:

Click here to view code image

```
class Widget {
public:
  ...

  static void* operator new(std::size_t size, std::ostream&
logStream)
    throw(std::bad_alloc);
  static void operator delete(void *pMemory) throw();
  static void operator delete(void *pMemory, std::ostream&
logStream)
    throw();
  ...
};
```

With this change, if an exception is thrown from the `Widget` constructor in this statement,

```
Widget *pw = new (std::cerr) Widget;   // as before, but no
leak this time
```

the corresponding placement `delete` is automatically invoked, and that allows `Widget` to ensure that no memory is leaked.

However, consider what happens if no exception is thrown (which will usually be the case) and we get to a `delete` in client code:

```
delete pw;                             // invokes the normal
                                       // operator delete
```

As the comment indicates, this calls the normal `operator delete`, not the placement version. Placement `delete` is called *only* if an exception arises from a constructor call that's coupled to a call to a placement `new`. Applying `delete` to a pointer (such as `pw` above) never yields a call to a placement version of `delete`. *Never.*

This means that to forestall all memory leaks associated with placement versions of `new`, you must provide both the normal `operator delete` (for when no exception is thrown during construction) and a placement version that takes the same extra arguments as `operator new` does (for when one is). Do that, and you'll never lose sleep over subtle memory leaks again. Well, at least not *these* subtle memory leaks.

Incidentally, because member function names hide functions with the same names in outer scopes (see Item 33), you need to be careful to avoid having class-specific `new`s hide other `new`s (including the normal versions) that your clients expect. For example, if you have a base class that declares only a placement version of `operator new`, clients will find that the normal form of `new` is unavailable to them:

Click here to view code image

```
class Base {
public:
  ...

  static void* operator new(std::size_t size,          //
this new hides
                            std::ostream& logStream)    //
the normal
    throw(std::bad_alloc);                              //
global forms
  ...
};

Base *pb = new Base;                        // error! the
normal form of
                                            // operator new
is hidden

Base *pb = new (std::cerr) Base;            // fine, calls
Base's
                                            // placement new
```

Similarly, `operator new`s in derived classes hide both global and inherited versions of `operator new`:

Click here to view code image

```
class Derived: public Base {                   // inherits
from Base above
public:
  ...

  static void* operator new(std::size_t size)  // redeclares
the normal
      throw(std::bad_alloc);                   // form of new
  ...
};
Derived *pd = new (std::clog) Derived;         // error!
Base's placement
                                               // new is
hidden

Derived *pd = new Derived;                     // fine, calls
Derived's
```

```
                                               // operator
new
```

Item 33 discusses this kind of name hiding in considerable detail, but for purposes of writing memory allocation functions, what you need to remember is that by default, C++ offers the following forms of `operator new` at global scope:

Click here to view code image

```
void* operator new(std::size_t)
throw(std::bad_alloc);      // normal new

void* operator new(std::size_t, void*)
throw();             // placement new

void* operator
new(std::size_t,                             // nothrow new —
                   const std::nothrow_t&)
throw();            // see Item 49
```

If you declare any `operator new`s in a class, you'll hide all these standard forms. Unless you mean to prevent class clients from using these forms, be sure to make them available in addition to any custom `operator new` forms you create. For each `operator new` you make available, of course, be sure to offer the corresponding `operator delete`, too. If you want these functions to behave in the usual way, just have your class-specific versions call the global versions.

An easy way to do this is to create a base class containing all the normal forms of `new` and `delete`:

Click here to view code image

```
class StandardNewDeleteForms {
public:
  // normal new/delete
  static void* operator new(std::size_t size)
throw(std::bad_alloc)
  { return ::operator new(size); }
  static void operator delete(void *pMemory) throw()
  { ::operator delete(pMemory); }
  // placement new/delete
  static void* operator new(std::size_t size, void *ptr)
throw()
  { return ::operator new(size, ptr); }
  static void operator delete(void *pMemory, void *ptr)
throw()
  { return ::operator delete(pMemory, ptr); }

  // nothrow new/delete
  static void* operator new(std::size_t size, const
std::nothrow_t& nt) throw()
  { return ::operator new(size, nt); }
  static void operator delete(void *pMemory, const
std::nothrow_t&) throw()
  { ::operator delete(pMemory); }
};
```

Clients who want to augment the standard forms with custom forms can then just use inheritance and `using` declarations (see Item 33) to get the standard forms:

Click here to view code image

```
class Widget: public StandardNewDeleteForms {           //
inherit std forms
public:
   using StandardNewDeleteForms::operator new;          //
make those
   using StandardNewDeleteForms::operator delete;       //
forms visible

   static void* operator new(std::size_t size,          //
add a custom
                             std::ostream& logStream)   //
placement new
     throw(std::bad_alloc);

   static void operator delete(void *pMemory,           //
add the corres
                               std::ostream& logStream) //
ponding place-
    throw();                                            //
ment delete
  ...
};
```

Things to Remember

- When you write a placement version of `operator new`, be sure to write the corresponding placement version of `operator delete`. If you don't, your program may experience subtle, intermittent memory leaks.
- When you declare placement versions of `new` and `delete`, be sure not to unintentionally hide the normal versions of those functions.

9. Miscellany

Welcome to the catch-all “Miscellany” chapter. There are only three Items here, but don't let their diminutive number or unglamorous setting fool you. They're important.

The first Item emphasizes that compiler warnings are not to be trifled with, at least not if you want your software to behave properly. The second offers an overview of the contents of the standard C++ library, including the significant new functionality being introduced in TR1. Finally, the last Item provides an overview of Boost, arguably the most important general-purpose C++-related web site. Trying to write effective C++ software without the information in these Items is, at best, an uphill battle.

Item 53: Pay attention to compiler warnings

Many programmers routinely ignore compiler warnings. After all, if the problem were serious, it would be an error, right? This thinking may be relatively harmless in other languages, but in C++, it's a good bet compiler writers have a better grasp of what's going on than you do. For example, here's an error everybody makes at one time or another:

```
class B {
public:
virtual void f() const;
};

class D: public B {
public:
virtual void f();
};
```

The idea is for `D::f` to redefine the virtual function `B::f`, but there's a mistake: in `B`, `f` is a `const` member function, but in `D` it's not declared `const`. One compiler I know says this about that:

```
warning: D::f() hides virtual B::f()
```

Too many inexperienced programmers respond to this message by saying to themselves, “Of *course* `D::f` hides `B::f` — that's what it's *supposed* to do!” Wrong. This compiler is trying to tell you that the `f` declared in `B` has not been redeclared in `D`; instead, it's been hidden entirely (see Item 33 for a description of why this is so). Ignoring this compiler warning will almost certainly lead to erroneous program behavior, followed by a lot of debugging to discover something this compiler detected in the first place.

After you gain experience with the warning messages from a particular compiler, you'll learn to understand what the different messages mean (which is often very different from what they *seem* to mean, alas). Once you have that experience, you may choose to ignore a whole range of warnings, though it's generally considered better practice to write code that compiles warning-free, even at the highest warning level. Regardless, it's important to make sure that before you dismiss a warning, you understand exactly what it's trying to tell you.

As long as we're on the topic of warnings, recall that warnings are inherently implementation-dependent, so it's not a good idea to get sloppy in your programming, relying on compilers to spot your mistakes for you. The function-hiding code above, for instance, goes through a different (but widely used) compiler with nary a squawk.

Things to Remember

- Take compiler warnings seriously, and strive to compile warning-free at the maximum warning level supported by your compilers.
- Don't become dependent on compiler warnings, because different compilers warn about different things. Porting to a new compiler may eliminate warning messages you've come to rely on.

Item 54: Familiarize yourself with the standard library, including TR1

The standard for C++ — the document defining the language and its library — was ratified in 1998. In 2003, a minor “bug-fix” update was issued. The standardization committee continues its work, however, and a “Version 2.0” C++ standard is expected around 2008 or so. The uncertainty regarding that date explains why people usually refer to the next version of C++ as “C++0x” — the 200x version of C++.

C++0x will probably include some interesting new language features, but most new C++ functionality will come in the form of additions to the standard library. We already know what some of the new library functionality will be, because it's been specified in a document known as TR1 (“Technical Report 1” from the C++ Library Working Group). The standardization committee reserves the right to modify TR1 functionality before it's officially enshrined in C++0x, but significant changes are unlikely. For all intents and purposes, TR1 heralds the beginning of a new release of C++ — what we might call standard C++ 1.1. You can't be an effective C++ programmer without being familiar with TR1 functionality, because that functionality is a boon to virtually every kind of library and application.

Before surveying what's in TR1, it's worth reviewing the major parts of the standard C++ library specified by C++98:

- **The Standard Template Library (STL)**, including containers (`vector`, `string`, `map`, etc.); iterators; algorithms (`find`, `sort`, `transform`, etc.); function objects (`less`, `greater`, etc.); and various container and function object adapters (`stack`, `priority_queue`, `mem_fun`, `not1`, etc.).
- **Iostreams**, including support for user-defined buffering, internationalized IO, and the predefined objects `cin`, `cout`, `cerr`, and `clog`.
- **Support for internationalization**, including the ability to have multiple active locales. Types like `wchar_t` (usually 16 bits/char) and `wstring` (`string`s of `wchar_t`s) facilitate working with Unicode.
- **Support for numeric processing**, including templates for complex numbers (`complex`) and arrays of pure values (`valarray`).
- **An exception hierarchy**, including the base class `exception`, its derived classes `logic_error` and `runtime_error`, and various classes that inherit from those.
- **C89's standard library**. Everything in the 1989 C standard library is also in C++.

If any of the above is unfamiliar to you, I suggest you schedule some quality time with your favorite C++ reference to rectify the situation.

TR1 specifies 14 new components (i.e., pieces of library functionality). All are in the `std` namespace, more precisely, in the nested namespace `tr1`. The full name of the TR1 component `shared_ptr` (see below) is thus `std::tr1::shared_ptr`. In this book, I customarily omit the `std::` when discussing components of the standard library, but I always prefix TR1 components with `tr1::`.

This book shows examples of the following TR1 components:

- The **smart pointers** `tr1::shared_ptr` and `tr1::weak_ptr`. `tr1::shared_ptr`s act like built-in pointers, but they keep track of how many `tr1::shared_ptr`s point to an object. This is known as *reference counting*. When the last such pointer is destroyed (i.e., when the reference count for an object becomes zero), the object is automatically deleted. This works well in preventing resource leaks in acyclic data structures, but if two or more objects contain `tr1::shared_ptr`s such that a cycle is formed, the cycle may keep each object's reference count above zero, even when all external pointers to the cycle have been destroyed (i.e., when the group of objects as a whole is unreachable). That's where `tr1::weak_ptr`s come in. `tr1::weak_ptr`s are designed to act as cycle-inducing pointers in otherwise acyclic `tr1::shared_ptr`-based data structures. `tr1::weak_ptr`s don't participate in reference counting. When the last `tr1::shared_ptr` to an object is destroyed, the object is deleted, even if `tr1::weak_ptr`s continue to point there. Such `tr1::weak_ptr`s are automatically marked as invalid, however.

  `tr1::shared_ptr` may be the most widely useful component in TR1. I use it many times in this book, including in Item 13, where I explain why it's so important. (The book contains no uses of `tr1::weak_ptr`, sorry.)

- **tr1::function**, which makes it possible to represent any *callable entity* (i.e., any function or function object) whose signature is consistent with a target signature. If we wanted to make it possible to register callback functions that take an `int` and return a `string`, we could do this:

Click here to view code image

```
void registerCallback(std::string func(int));    // param
type is a function
                                                 // taking an
int and
                                                 // returning
a string
```

The parameter name `func` is optional, so `registerCallback` could be declared this way, instead:

Click here to view code image

```
void registerCallback(std::string (int));          // same as
above; param
                                                   // name is
omitted
```

Note here that "`std::string (int)`" is a function signature.

`tr1::function` makes it possible to make `registerCallback` much more flexible, accepting as its argument any callable entity that takes an `int` or *anything an* `int` can be converted into and that returns a `string` or *anything convertible to a* `string`. `tr1::function` takes as a template parameter its target function signature:

Click here to view code image

```
void registerCallback(std::tr1::function<std::string (int)>
func);
                                                    // the param
"func" will
                                                    // take any
callable entity
                                                    // with a
sig consistent
                                                    // with
"std::string (int)"
```

This kind of flexibility is astonishingly useful, something I do my best to demonstrate in Item 35.

- **tr1::bind**, which does everything the STL binders `bind1st` and `bind2nd` do, plus much more. Unlike the pre-TR1 binders, `tr1::bind` works with both `const` and non-`const` member functions. Unlike the pre-TR1 binders, `tr1::bind` works with by-reference parameters. Unlike the pre-TR1 binders, `tr1::bind` handles function pointers without help, so there's no need to mess with `ptr_fun`, `mem_fun`, or `mem_fun_ref` before calling `tr1::bind`. Simply put, `tr1::bind` is a second-generation binding facility that is significantly better than its predecessor. I show an example of its use in Item 35.

I divide the remaining TR1 components into two sets. The first group offers fairly discrete standalone functionality:

- **Hash tables** used to implement sets, multisets, maps, and multimaps. Each new container has an interface modeled on that of its pre-TR1 counterpart. The most surprising thing about TR1's hash tables are their names: `tr1::unordered_set`, `tr1::unordered_multiset`, `tr1::unordered_map`, and `tr1::unordered_multimap`. These names emphasize that, unlike the contents of a `set`, `multiset`, `map`, or `multimap`, the elements in a TR1 hash-based container are not in any predictable order.
- **Regular expressions**, including the ability to do regular expression-based search and replace operations on strings, to iterate through strings from match to match, etc.
- **Tuples**, a nifty generalization of the `pair` template that's already in the standard library. Whereas `pair` objects can hold only two objects, however, `tr1::tuple` objects can hold an arbitrary number. Expat Python and Eiffel programmers, rejoice! A little piece of your former homeland is now part of C++.
- **tr1::array**, essentially an “STLified” array, i.e., an array supporting member functions like `begin` and `end`. The size of a `tr1::array` is fixed during compilation; the object uses no dynamic memory.
- **tr1::mem_fn**, a syntactically uniform way of adapting member function pointers. Just as `tr1::bind` subsumes and extends the capabilities of C++98's `bind1st` and `bind2nd`, `tr1::mem_fn` subsumes and extends the capabilities of C++98's `mem_fun` and `mem_fun_ref`.
- **tr1::reference_wrapper**, a facility to make references act a bit more like objects. Among other things, this makes it possible to create containers that act as if they hold references. (In reality, containers can hold only objects or pointers.)
- **Random number generation** facilities that are vastly superior to the `rand` function that C++ inherited from C's standard library.
- **Mathematical special functions**, including Laguerre polynomials, Bessel functions, complete elliptic integrals, and many more.
- **C99 compatibility extensions**, a collection of functions and templates designed to bring many new C99 library features to C++.

The second set of TR1 components consists of support technology for more sophisticated template programming techniques, including template metaprogramming (see Item 48):

- **Type traits**, a set of traits classes (see Item 47) to provide compile-time information about types. Given a type `T`, TR1's type traits can reveal whether `T` is a built-in type, offers a virtual destructor, is an empty class (see Item 39), is implicitly convertible to some other type `U`, and much more. TR1's type traits can also reveal the proper alignment for a type, a crucial piece of information for programmers writing custom memory allocation functions (see Item 50).
- **tr1::result_of**, a template to deduce the return types of function calls. When writing templates, it's often important to be able to refer to the type of object returned from a call to a function (template), but the return type can depend on the function's parameter types in complex ways. `tr1::result_of` makes referring to function return types easy. `tr1::result_of` is used in several places in TR1 itself.

Although the capabilities of some pieces of TR1 (notably `tr1::bind` and `tr1::mem_fn`) subsume those of some pre-TR1 components, TR1 is a pure addition to the standard library. No TR1 component replaces an existing component, so legacy code written with pre-TR1 constructs continues to be valid.

TR1 itself is just a document.[†] To take advantage of the functionality it specifies, you need access to code that implements it. Eventually, that code will come bundled with compilers, but as I write this in 2005, there is a good chance that if you look for TR1 components in your standard library implementations, at least some will be missing. Fortunately, there is someplace else to look: 10 of the 14 components in TR1 are based on libraries freely available from Boost (see Item 55), so that's an excellent resource for TR1-like functionality. I say “TR1-like,” because, though much TR1 functionality is based on Boost libraries, there are places where Boost functionality is currently not an exact match for the TR1 specification. It's possible that by the time you read this, Boost not only will have TR1-conformant implementations for the TR1 components that evolved from Boost libraries, it will also offer implementations of the four TR1 components that were not based on Boost work.

If you'd like to use Boost's TR1-like libraries as a stopgap until compilers ship with their own TR1 implementations, you may want to avail yourself of a namespace trick. All Boost components are in the namespace `boost`, but TR1 components are supposed to be in `std::tr1`. You can tell your compilers to treat references to `std::tr1` the same as references to `boost`. This is how:

Click here to view code image

```
namespace std {
namespace tr1 = ::boost;           // namespace std::tr1 is
an alias
}                                   // for namespace boost
```

Technically, this puts you in the realm of undefined behavior, because, as Item 25 explains, you're not allowed to add anything to the `std` namespace. In practice, you're unlikely to run into any trouble. When your compilers provide their own TR1 implementations, all you'll need to do is eliminate the above namespace alias; code referring to `std::tr1` will continue to be valid.

Probably the most important part of TR1 not based on Boost libraries is hash tables, but hash tables have been available for many years from several sources under the names `hash_set`, `hash_multiset`, `hash_map`, and `hash_multimap`. There is a good chance that the libraries shipping with your compilers already contain these templates. If not, fire up your favorite search engine and search for these names (as well as their TR1 appellations), because you're sure to find several sources for them, both commercial and freeware.

Things to Remember

- The primary standard C++ library functionality consists of the STL, iostreams, and locales. The C89 standard library is also included.
- TR1 adds support for smart pointers (e.g., `tr1::shared_ptr`), generalized function pointers (`tr1::function`), hash-based containers, regular expressions, and 10 other components.
- TR1 itself is only a specification. To take advantage of TR1, you need an implementation. One source for implementations of TR1 components is Boost.

Item 55: Familiarize yourself with Boost.

Searching for a collection of high-quality, open source, platform- and compiler-independent libraries? Look to Boost. Interested in joining a community of ambitious, talented C++ developers working on state-of-the-art library design and implementation? Look to Boost. Want a glimpse of what C++ might look like in the future? Look to Boost.

Boost is both a community of C++ developers and a collection of freely downloadable C++ libraries. Its web site is http://boost.org. You should bookmark it now.

There are many C++ organizations and web sites, of course, but Boost has two things going for it that no other organization can match. First, it has a uniquely close and influential relationship with the C++ standardization committee. Boost was founded by committee members, and there continues to be strong overlap between the Boost and committee memberships. In addition, Boost has always had as one of its goals to act as a testing ground for capabilities that could be added to Standard C++. One result of this relationship is that of the 14 new libraries introduced into C++ by TR1 (see Item 54), more than two-thirds are based on work done at Boost.

The second special characteristic of Boost is its process for accepting libraries. It's based on public peer review. If you'd like to contribute a library to Boost, you start by posting to the Boost developers mailing list to gauge interest in the library and initiate the process of preliminary examination of your work. Thus begins a cycle that the web site summarizes as “Discuss, refine, resubmit. Repeat until satisfied.”

Eventually, you decide that your library is ready for formal submission. A review manager confirms that your library meets Boost's minimal requirements. For example, it must compile under at least two compilers (to demonstrate nominal portability), and you have to attest that the library can be made available under an acceptable license (e.g., the library must allow free commercial and non-commercial use). Then your submission is made available to the Boost community for official review. During the review period, volunteers go over your library materials (e.g., source code, design documents, user documentation, etc.) and consider questions such as these:

- How good are the design and implementation?
- Is the code portable across compilers and operating systems?
- Is the library likely to be of use to its target audience, i.e., people working in the domain the library addresses?

- Is the documentation clear, complete, and accurate?

These comments are posted to a Boost mailing list, so reviewers and others can see and respond to one another's remarks. At the end of the review period, the review manager decides whether your library is accepted, conditionally accepted, or rejected.

Peer reviews do a good job of keeping poorly written libraries out of Boost, but they also help educate library authors in the considerations that go into the design, implementation, and documentation of industrial-strength cross-platform libraries. Many libraries require more than one official review before being declared worthy of acceptance.

Boost contains dozens of libraries, and more are added on a continuing basis. From time to time, some libraries are also removed, typically because their functionality has been superseded by a newer library that offers greater functionality or a better design (e.g., one that is more flexible or more efficient).

The libraries vary widely in size and scope. At one extreme are libraries that conceptually require only a few lines of code (but are typically much longer after support for error handling and portability is added). One such library is **Conversion**, which provides safer or more convenient cast operators. Its `numeric_cast` function, for example, throws an exception if converting a numeric value from one type to another leads to overflow or underflow or a similar problem, and `lexical_cast` makes it possible to cast any type supporting `operator<<` into a string — very useful for diagnostics, logging, etc. At the other extreme are libraries offering such extensive capabilities, entire books have been written about them. These include the **Boost Graph Library** (for programming with arbitrary graph structures) and the **Boost MPL Library** ("metaprogramming library").

Boost's bevy of libraries addresses a cornucopia of topics, grouped into over a dozen general categories. Those categories include:

- **String and text processing**, including libraries for type-safe `printf`-like formatting, regular expressions (the basis for similar functionality in TR1 — see Item 54), and tokenizing and parsing.
- **Containers**, including libraries for fixed-size arrays with an STL-like interface (see Item 54), variable-sized bitsets, and multidimensional arrays.

- **Function objects and higher-order programming**, including several libraries that were used as the basis for functionality in TR1. One interesting library is the Lambda library, which makes it so easy to create function objects on the fly, you're unlikely to realize that's what you're doing:

Click here to view code image

```
    using namespace boost::lambda;                    //
    make boost::lambda
                                                      //
    functionality visible

    std::vector<int> v;

    ...

    std::for_each(v.begin(), v.end(),                 // for
    each element x in
                  std::cout << _1 * 2 + 10 << "\n");  // v,
    print x*2+10;
                                                      //
    "_1" is the Lambda
                                                      //
    library's placeholder
                                                      // for
    the current element
```

- **Generic programming**, including an extensive set of traits classes. (See Item 47 for information on traits).
- **Template metaprogramming** (TMP — see Item 48), including a library for compile-time assertions, as well as the Boost MPL Library. Among the nifty things in MPL is support for STL-like data structures of compile-time entities like *types*, e.g.,

Click here to view code image

```
// create a list-like compile-time container of three types
(float,
// double, and long double) and call the container "floats"
typedef boost::mpl::list<float, double, long double> floats;

// create a new compile-time list of types consisting of the
```

```
types in
// "floats" plus "int" inserted at the front; call the new
container "types"
typedef boost::mpl::push_front<floats, int>::type types;
```

Such containers of types (often known as *typelists*, though they can also be based on an `mpl::vector` as well as an `mpl::list`) open the door to a wide range of powerful and important TMP applications.

- **Math and numerics**, including libraries for rational numbers; octonions and quaternions; greatest common divisor and least common multiple computations; and random numbers (yet another library that influenced related functionality in TR1).
- **Correctness and testing**, including libraries for formalizing implicit template interfaces (see Item 41) and for facilitating test-first programming.
- **Data structures**, including libraries for type-safe unions (i.e., storing variant “any” types) and the tuple library that led to the corresponding TR1 functionality.
- **Inter-language support**, including a library to allow seamless interoperability between C++ and Python.
- **Memory**, including the Pool library for high-performance fixed-size allocators (see Item 50); and a variety of smart pointers (see Item 13), including (but not limited to) the smart pointers in TR1. One such non-TR1 smart pointer is `scoped_array`, an `auto_ptr`-like smart pointer for dynamically allocated arrays; Item 44 shows an example use.
- **Miscellaneous**, including libraries for CRC checking, date and time manipulations, and traversing file systems.

Remember, that's just a sampling of the libraries you'll find at Boost. It's not an exhaustive list.

Boost offers libraries that do many things, but it doesn't cover the entire programming landscape. For example, there is no library for GUI development, nor is there one for communicating with databases. At least there's not now — not as I write this. By the time you read it, however, there might be. The only way to know for sure is to check. I suggest you do it right now: http://boost.org. Even if you don't find exactly what you're looking for, you're certain to find something interesting there.

Things to Remember

- Boost is a community and web site for the development of free, open source, peer-reviewed C++ libraries. Boost plays an influential role in C++ standardization.
- Boost offers implementations of many TR1 components, but it also offers many other libraries, too.

A. Beyond *Effective C++*

Effective C++ covers what I consider to be the most important general guidelines for practicing C++ programmers, but if you're interested in more ways to improve your effectiveness, I encourage you to examine my other C++ books, *More Effective C++* and *Effective STL*.

More Effective C++ covers additional programming guidelines and includes extensive treatments of topics such as efficiency and programming with exceptions. It also describes important C++ programming techniques like smart pointers, reference counting, and proxy objects.

Effective STL is a guideline-oriented book like *Effective C++*, but it focuses exclusively on making effective use of the Standard Template Library.

Tables of contents for both books are summarized below.

Contents of *More Effective C++*

Basics

- Item 1: Distinguish between pointers and references
- Item 2: Prefer C++-style casts
- Item 3: Never treat arrays polymorphically
- Item 4: Avoid gratuitous default constructors

Operators

- Item 5: Be wary of user-defined conversion functions
- Item 6: Distinguish between prefix and postfix forms of increment and decrement operators
- Item 7: Never overload `&&`, `||`, or `,`
- Item 8: Understand the different meanings of `new` and `delete`

Exceptions

- Item 9: Use destructors to prevent resource leaks
- Item 10: Prevent resource leaks in constructors
- Item 11: Prevent exceptions from leaving destructors
- Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function

Item 13: Catch exceptions by reference
Item 14: Use exception specifications judiciously
Item 15: Understand the costs of exception handling

Efficiency

Item 16: Remember the 80-20 rule
Item 17: Consider using lazy evaluation
Item 18: Amortize the cost of expected computations
Item 19: Understand the origin of temporary objects
Item 20: Facilitate the return value optimization
Item 21: Overload to avoid implicit type conversions
Item 22: Consider using *op*= instead of stand-alone *op*
Item 23: Consider alternative libraries
Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI

Techniques

Item 25: Virtualizing constructors and non-member functions
Item 26: Limiting the number of objects of a class
Item 27: Requiring or prohibiting heap-based objects
Item 28: Smart pointers
Item 29: Reference counting
Item 30: Proxy classes
Item 31: Making functions virtual with respect to more than one object

Miscellany

Item 32: Program in the future tense
Item 33: Make non-leaf classes abstract
Item 34: Understand how to combine C++ and C in the same program
Item 35: Familiarize yourself with the language standard

Contents of *Effective STL*

Chapter 1: Containers

Item 1: Choose your containers with care.

Item 2: Beware the illusion of container-independent code.
Item 3: Make copying cheap and correct for objects in containers.
Item 4: Call `empty` instead of checking `size()` against zero.
Item 5: Prefer range member functions to their single-element counterparts.
Item 6: Be alert for C++'s most vexing parse.
Item 7: When using containers of `new`ed pointers, remember to `delete` the pointers before the container is destroyed.
Item 8: Never create containers of `auto_ptr`s.
Item 9: Choose carefully among erasing options.
Item 10: Be aware of allocator conventions and restrictions.
Item 11: Understand the legitimate uses of custom allocators.
Item 12: Have realistic expectations about the thread safety of STL containers.
Chapter 2: `vector` and `string`
Item 13: Prefer `vector` and `string` to dynamically allocated arrays.
Item 14: Use `reserve` to avoid unnecessary reallocations.
Item 15: Be aware of variations in `string` implementations.
Item 16: Know how to pass `vector` and `string` data to legacy APIs.
Item 17: Use “the `swap` trick” to trim excess capacity.
Item 18: Avoid using `vector<bool>`.
Chapter 3: Associative Containers
Item 19: Understand the difference between equality and equivalence.
Item 20: Specify comparison types for associative containers of pointers.
Item 21: Always have comparison functions return `false` for equal values.
Item 22: Avoid in-place key modification in `set` and `multiset`.
Item 23: Consider replacing associative containers with sorted `vector`s.
Item 24: Choose carefully between `map::operator[]` and `map::insert` when efficiency is important.

Item 25: Familiarize yourself with the nonstandard hashed containers.
Chapter 4: Iterators
Item 26: Prefer `iterator` to `const_iterator`, `reverse_iterator`, and `const_reverse_iterator`.
Item 27: Use `distance` and `advance` to convert a container's `const_iterator`s to `iterator`s.
Item 28: Understand how to use a `reverse_iterator`'s base `iterator`.
Item 29: Consider `istreambuf_iterator`s for character-by-character input.
Chapter 5: Algorithms
Item 30: Make sure destination ranges are big enough.
Item 31: Know your sorting options.
Item 32: Follow `remove`-like algorithms by `erase` if you really want to remove something.
Item 33: Be wary of `remove`-like algorithms on containers of pointers.
Item 34: Note which algorithms expect sorted ranges.
Item 35: Implement simple case-insensitive string comparisons via `mismatch` or `lexicographical_compare`.
Item 36: Understand the proper implementation of `copy_if`.
Item 37: Use `accumulate` or `for_each` to summarize ranges.
Chapter 6: Functors, Functor Classes, Functions, etc.
Item 38: Design functor classes for pass-by-value.
Item 39: Make predicates pure functions.
Item 40: Make functor classes adaptable.
Item 41: Understand the reasons for `ptr_fun`, `mem_fun`, and `mem_fun_ref`.
Item 42: Make sure `less<T>` means `operator<`.
Chapter 7: Programming with the STL
Item 43: Prefer algorithm calls to hand-written loops.
Item 44: Prefer member functions to algorithms with the same names.
Item 45: Distinguish among `count`, `find`, `binary_search`, `lower_bound`, `upper_bound`, and `equal_range`.
Item 46: Consider function objects instead of functions as algorithm

parameters.
Item 47: Avoid producing write-only code.
Item 48: Always `#include` the proper headers.
Item 49: Learn to decipher STL-related compiler diagnostics.
Item 50: Familiarize yourself with STL-related web sites.

B. Item Mappings Between Second and Third Editions

This third edition of *Effective C++* differs from the second edition in many ways, most significantly in that it includes lots of new information. However, most of the second edition's content remains in the third edition, albeit often in a modified form and location. In the tables on the pages that follow, I show where information in second edition Items may be found in the third edition and vice versa.

The tables show a mapping of *information*, not text. For example, the ideas in Item 39 of the second edition (“Avoid casts down the inheritance hierarchy”) are now found in Item 27 of the current edition (“Minimize casting”), even though the third edition text and examples for that Item are entirely new. A more extreme example involves the second edition's Item 18 (“Strive for class interfaces that are complete and minimal”). One of the primary conclusions of that Item was that prospective member functions that need no special access to the non-public parts of the class should generally be non-members. In the third edition, that same result is reached via different (stronger) reasoning, so Item 18 in the second edition maps to Item 23 in the third edition (“Prefer non-member non-friend functions to member functions”), even though about the only thing the two Items have in common is their conclusion.

Second Edition to Third Edition

2nd Ed.	3rd Ed.	2nd Ed.	3rd Ed.	2nd Ed.	3rd Ed.
1	2	18	23	35	32
2	—	19	24	36	34
3	—	20	22	37	36
4	—	21	3	38	37
5	16	22	20	39	27
6	13	23	21	40	38
7	49	24	—	41	41
8	51	25	—	42	39, 44
9	52	26	—	43	40
10	50	27	6	44	—
11	14	28	—	45	5
12	4	29	28	46	18
13	4	30	28	47	4
14	7	31	21	48	53
15	10	32	26	49	54
16	12	33	30	50	—
17	11	34	31		

Third Edition to Second Edition

3rd Ed.	2nd Ed.	3rd Ed.	2nd Ed.	3rd Ed.	2nd Ed.
1	—	20	22	39	42
2	1	21	23, 31	40	43
3	21	22	20	41	41
4	12, 13, 47	23	18	42	—
5	45	24	19	43	—
6	27	25	—	44	42
7	14	26	32	45	—
8	—	27	39	46	—
9	—	28	29, 30	47	—
10	15	29	—	48	—
11	17	30	33	49	7
12	16	31	34	50	10
13	6	32	35	51	8
14	11	33	9	52	9
15	—	34	36	53	48
16	5	35	—	54	49
17	—	36	37	55	—
18	46	37	38		
19	pp. 77–79	38	40		

Index

Operators are listed under *operator*. That is, `operator<<` is listed under `operator<<`, not under `<<`, etc.

Example classes, structs, and class or struct templates are indexed under *example classes/templates*. Example function and function templates are indexed under *example functions/templates*.

Before A

.NET 7A, 81A, 135A, 145A, 194A
    see also C#
=, in initialization vs. assignment 6A
1066 150A
2nd edition of this book
    compared to 3rd edition, 277–279A
3rd edition of this book
    compared to 2nd edition, 277–279A
80-20 rule, 139A, 168A

A

abstract classes, 43A
accessibility
    control over data members', 95A
    name, multiple inheritance and, 193A
accessing names, in templatized bases, 207–212A
addresses
    inline functions, 136A
    objects, 118A
aggregation, see composition
aliasing, 54A
alignment, 249–250A
allocators, in the STL, 240A

alternatives to virtual functions, 169–177A
ambiguity
  multiple inheritance and, 192A
  nested dependent names and types, 205A
argument-dependent lookup, 110A
arithmetic, mixed-mode, 103A, 222–226A
array layout, vs. object layout, 73A
array `new` 254–255AA
array, invalid index and, 7A
`ASPECT_RATIO`, 13A
assignment
  see also `operator=`
  chaining assignments, 52A
  copy-and-`swap` and, 56A
  generalized, 220A
  to self, `operator=` and, 53–57A
  vs. initialization, 6A, 27–29A, 114A
assignment operator, copy, 5A
`auto_ptr`, see `std::auto_ptr`
automatically generated functions, 34–37A
  copy constructor and copy assignment operator, 221A
  disallowing, 37–39A
avoiding code duplication, 50A, 60A

B

Barry, Dave, allusion to, 229A
base classes
  copying, 59A
  duplication of data in, 193A
  lookup in, `this->` and, 210A
  names hidden in derived classes, 263A
  polymorphic, 44A
  polymorphic, destructors and, 40–44A

- templatized, 207–212A
- virtual, 193A

basic guarantee, the, 128A

Battle of Hastings, 150A

bidirectional iterators, 227A

`bidirectional_iterator_tag`, 228A

binary upgradeability, inlining and, 138A

binding
- dynamic, see dynamic binding
- static, see static binding

birds and penguins, 151–153A

bitwise const member functions, 21–22A

books
- *Effective STL* , 273A, 275–276A
- *More Effective C++* , 273A, 273–274A
- *Some Must Watch While Some Must Sleep*, 150A

Boost, 10A, 269–272A
- containers, 271A
- Conversion library, 270A
- correctness and testing support, 272A
- data structures, 272A
- function objects and higher-order programming utilities, 271A
- functionality not provided, 272A
- generic programming support, 271A
- Graph library, 270A
- inter-language support, 272A
- Lambda library, 271A
- math and numerics utilities, 271A
- memory management utilities, 272A
- MPL library, 270A, 271A

`noncopyable` base class, 39A
- Pool library, 250A, 251A

`scoped_array`, 65A, 216A, 272A

`shared_array`, 65A
`shared_ptr` implementation, costs, 83A
    smart pointers, 65A, 272A
    string and text utilities, 271A
    template metaprogramming support, 271A
    TR1 and, 9–10A, 268A, 269A
    typelist support, 271A
    web site, 10A, 269A, 272
`boost`, as synonym for `std::tr1`, 268A
breakpoints, and inlining, 139A
built-in types, 26–27A
    efficiency and passing, 89A
    incompatibilities with, 80A

C

C standard library and C++ standard library, 264A
C#, 43A, 76A, 97A, 100A, 116A, 118A, 190A
    see also .NET
C++ standard library, 263–269A
`<iosfwd>` and, 144A
    array replacements and, 75A
    C standard library and, 264A
    C89 standard library and, 264A
    header organization of, 101A
`list` template, 186A
`logic_error` and, 113A
`set` template, 185A
`vector` template, 75A
C++, as language federation, 11–13A
C++0x, 264A
C++-style casts, 117A
C, as sublanguage of C++, 12A
C99 standard library, TR1 and, 267A

caching
`const` and, 22A
    `mutable` and, 22A
calling `swap`, 110A
calls to base classes, casting and, 119A
casting, 116–123A
    see also `const_cast`, `static_cast`, `dynamic_cast`, and `reinterpret_cast`
    base class calls and, 119A
    constness away, 24–25A
    encapsulation and, 123A
`grep` and, 117A
    syntactic forms, 116–117A
    type systems and, 116A
    undefined behavior and, 119A
chaining assignments, 52A
class definitions
    artificial client dependencies, eliminating, 143A
    class declarations vs., 143A
    object sizes and, 141A
class design, see type design
class names, explicitly specifying, 162A
`class`, vs. `typename`, 203A
classes
    see also class definitions, interfaces
    abstract, 43A, 162A
    base
        see also base classes
        duplication of data in, 193A
        polymorphic, 44A
        templatized, 207–212A
        virtual, 193A
    defining, 4A
    derived

        - see also inheritance
        - virtual base initialization of, 194A
    - Handle, 144–145A
    - Interface, 145–147A
    - meaning of no virtual functions, 41A
    - RAII, see RAII
    - specification, see interfaces
    - traits, 226–232A
- client, 7A
- clustering objects, 251A
- code
    - bloat, 24A, 135A, 230A
        - avoiding, in templates, 212–217A
    - copy assignment operator, 60A
    - duplication, see duplication
    - exception-safe, 127–134A
    - factoring out of templates, 212–217A
    - incorrect, efficiency and, 90A
    - reuse, 195A
    - sharing, see duplication, avoiding
- common features and inheritance, 164A
- commonality and variability analysis, 212A
- compatibility, `vptr`s and, 42A
- compatible types, accepting, 218–222A
- compilation dependencies, 140–148A
    - minimizing, 140–148A, 190A
    - pointers, references, and objects and, 143A
- compiler warnings, 262–263A
    - calls to virtuals and, 50A
    - inlining and, 136A
    - partial copies and, 58A
- compiler-generated functions, 34–37A
    - disallowing, 37–39A

  - functions compilers may generate, 221A
- compilers
  - parsing nested dependent names, 204A
  - programs executing within, see template metaprogramming
  - register usage and, 89A
  - reordering operations, 76A
- `typename` and, 207A
  - when errors are diagnosed, 212A
- compile-time polymorphism, 201A
- composition, 184–186A
  - meanings of, 184A
  - replacing private inheritance with, 189A
  - synonyms for, 184A
  - vs. private inheritance, 188A
- conceptual constness, see `const`, logical
- consistency with the built-in types, 19A, 86A
- `const`, 13A, 17–26A
  - bitwise, 21–22A
  - caching and, 22A
  - casting away, 24–25A
  - function declarations and, 18A
  - logical, 22–23A
  - member functions, 19–25A
  - duplication and, 23–25A
  - members, initialization of, 29A
  - overloading on, 19–20A
  - pass by reference and, 86–90A
  - passing `std::auto_ptr` and, 220A
  - pointers, 17A
  - return value, 18A
  - uses, 17A
  - vs. `#define` 13–14A
- `const_cast`, 25A, 117A

- see also casting

`const_iterator`, vs. `iterator`s, 18A

constants, see `const`

constraints on interfaces, from inheritance, 85A

constructors, 84A
    - copy, 5A
    - default, 4A
    - empty, illusion of, 137A

`explicit`, 5A, 85A, 104A
    - implicitly generated, 34A
    - inlining and, 137–138A

`operator new` and, 137A
    - possible implementation in derived classes, 138A
    - relationship to `new`, 73A
    - static functions and, 52A
    - virtual, 146A, 147A
    - virtual functions and, 48–52A
    - with vs. without arguments, 114A

containers, in Boost, 271A

containment, see composition

`continue`, `delete` and, 62A

control over data members' accessibility, 95A

convenience functions, 100A

Conversion library, in Boost, 270A

conversions, type, see type conversions

copies, partial, 58A

copy assignment operator, 5A
    - code in copy constructor and, 60A
    - derived classes and, 60A

copy constructors
    - default definition, 35A
    - derived classes and, 60A
    - generalized, 219A

- how used, 5A
- implicitly generated, 34A
- pass-by-value and, 6A

copy-and-`swap`, 131A
- assignment and, 56A
- exception-safe code and, 132A

copying
- base class parts, 59A
- behavior, resource management and, 66–69A
- functions, the, 57A
- objects, 57–60A

correctness
- designing interfaces for, 78–83A
- testing and, Boost support, 272A

corresponding forms of `new` and `delete`, 73–75A

corrupt data structures, exception-safe code and, 127A

cows, coming home, 139A

crimes against English, 39A, 204A

cross-DLL problem, 82A

CRTP, 246A

C-style casts, 116A

ctor, 8A

curiously recurring template pattern, 246A

D

dangling handles, 126A

data members
- adding, copying functions and, 58A
- control over accessibility, 95A
- protected, 97A
- static, initialization of, 242A
- why private, 94–98A

data structures

  - exception-safe code and, 127A
  - in Boost, 272A
- deadly MI diamond, 193A
- debuggers
- `#define` and, 13A
  - inline functions and, 139A
- declarations, 3A
  - inline functions, 135A
  - replacing definitions, 143A
  - static const integral members, 14A
- default constructors, 4A
  - construction with arguments vs., 114A
  - implicitly generated, 34A
- default implementations
  - for virtual functions, danger of, 163–167A
  - of copy constructor, 35A
  - of `operator=`, 35A
- default initialization, unintended, 59A
- default parameters, 180–183A
  - impact if changed, 183A
  - static binding of, 182A
- `#define`
- debuggers and, 13A
  - disadvantages of, 13A, 16A
  - vs. `const` 13–14A
  - vs. inline functions, 16–17A
- definitions, 4A
  - classes, 4A
  - deliberate omission of, 38A
  - functions, 4A
  - implicitly generated functions, 35A
  - objects, 4A
  - pure virtual functions, 162A, 166–167A
  - replacing with declarations, 143A

  static class members, 242A
  static const integral members, 14A
  templates, 4A
  variable, postponing, 113–116A
`delete` see also `operator delete`
  forms of, 73–75A
  `operator delete` and, 73A
  relationship to destructors, 73A
  usage problem scenarios, 62A
`delete []`, `std::auto_ptr` and `tr1::shared_ptr` and, 65A
deleters
`std::auto_ptr` and, 68A
`tr1::shared_ptr` and, 68A, 81–83A
Delphi, 97A
Dement, William, 150A
dependencies, compilation, 140–148A
dependent names, 204A
dereferencing a null pointer, undefined behavior of, 6A
derived classes
  copy assignment operators and, 60A
  copy constructors and, 60A
  hiding names in base classes, 263A
  implementing constructors in, 138A
  virtual base initialization and, 194A
design
  contradiction in, 179A
  of interfaces, 78–83A
  of types, 78–86A
design patterns
  curiously recurring template (CRTP), 246A
  encapsulation and, 173A
  generating from templates, 237A
  Singleton, 31A

- Strategy, 171–177A
- Template Method, 170A
- TMP and, 237A

destructors, 84A
- exceptions and, 44–48A
- inlining and, 137–138A
- pure virtual, 43A
- relationship to `delete`, 73A
- resource managing objects and, 63A
- static functions and, 52A
- virtual `operator delete` and, 255A
- polymorphic base classes and, 40–44A
- virtual functions and, 48–52A

dimensional unit correctness, TMP and, 236A

DLLs, `delete` and, 82A

dtor, 8A

duplication
- avoiding, 23–25A, 29A, 50A, 60A, 164A, 183A, 212–217A
- base class data and, 193A

`init` function and, 60A

dynamic binding
- definition of, 181A
- of virtual functions, 179A

dynamic type, definition of, 181A

`dynamic_cast`, 50A, 117A, 120–123A
- see also casting
- efficiency of, 120A

E

early binding, 180A

easy to use correctly and hard to use incorrectly, 78–83A

EBO, see empty base optimization *Effective C++*, compared to *More Effective C++* and *Effective STL*, 273A

- *Effective STL*, 273A, 275–276A
  - compared to *Effective C++*, 273A
  - contents of, 275–276A
- efficiency
  - assignment vs. construction and destruction, 94A
  - default parameter binding, 182A
- `dynamic_cast`, 120A
  - Handle classes, 147A
  - incorrect code and, 90A, 94A
  - init. with vs. without args, 114A
  - Interface classes, 147A
  - macros vs. inline functions, 16A
  - member init. vs. assignment, 28A
  - minimizing compilation dependencies, 147A
- `operator new`/`operator delete` and, 248A
  - pass-by-reference and, 87A
  - pass-by-value and, 86–87A
  - passing built-in types and, 89A
  - runtime vs. compile-time tests, 230A
  - template metaprogramming and, 233A
  - template vs. function parameters, 216A
  - unused objects, 113A
  - virtual functions, 168A
- Eiffel, 100A
- embedding, see composition
- empty base optimization (EBO), 190–191A
- encapsulation, 95A, 99A
  - casts and, 123A
  - design patterns and, 173A
  - handles and, 124A
  - measuring, 99A
  - protected members and, 97A
  - RAII classes and, 72A

enum hack, 15–16A, 236A
errors
    detected during linking, 39A, 44A
    runtime, 152A
evaluation order, of parameters, 76A
example classes/templates
`A`, 4A
`ABEntry`, 27A
`AccessLevels`, 95A
`Address`, 184A
`Airplane`, 164A, 165A, 166A
`Airport`, 164A
`AtomicClock`, 40A
`AWOV`, 43A
`B`, 4A, 178A, 262A
`Base`, 54A, 118A, 137A, 157A, 158A, 159A, 160A, 254A, 255A, 259A
`BelowBottom`, 219A
`bidirectional_iterator_tag`, 228A
`Bird`, 151A, 152A, 153A
`Bitmap`, 54A
`BorrowableItem`, 192A
`Bottom`, 218A
`BuyTransaction`, 49A, 51A
`C`, 5A
`Circle`, 181A
`CompanyA`, 208A
`CompanyB`, 208A
`CompanyZ`, 209A
`CostEstimate`, 15A
`CPerson`, 198A
`CTextBlock`, 21A, 22A, 23A
`Customer`, 57A, 58A
`D`, 178A, 262A

`DatabaseID`, 197A

`Date`, 58A, 79A

`Day`, 79A

`DBConn`, 45A, 47A

`DBConnection`, 45A

`deque`, 229A

`deque::iterator`, 229A

`Derived`, 54A, 118A, 137A, 157A, 158A, 159A, 160A, 206A, 254A, 260A

`Directory`, 31A

`ElectronicGadget`, 192A

`Ellipse`, 161A

`Empty`, 34A, 190A

`EvilBadGuy`, 172A, 174A

`EyeCandyCharacter`, 175A

`Factorial`, 235A

`Factorial<0>`, 235A

`File`, 193A, 194A

`FileSystem`, 30A

`FlyingBird`, 152A

`Font`, 71A

`forward_iterator_tag`, 228A

`GameCharacter`, 169A, 170A, 172A, 173A, 176A

`GameLevel`, 174A

`GamePlayer`, 14A, 15A

`GraphNode`, 4A

`GUIObject`, 126A

`HealthCalcFunc`, 176A

`HealthCalculator`, 174A

`HoldsAnInt`, 190A, 191A

`HomeForSale`, 37A, 38A, 39A

`input_iterator_tag`, 228A

`input_iterator_tag<Iter*>`, 230A

`InputFile`, 193A, 194A

`Investment`, 61A, 70A
`IOFile`, 193A, 194A
`IPerson`, 195A, 197A
`iterator_traits`, 229A
    see also *std::iterator_traits*
`list`, 229A
`list::iterator`, 229A
`Lock`, 66A, 67A, 68A
`LoggingMsgSender`, 208A, 210A, 211A
`Middle`, 218A
`ModelA`, 164A, 165A, 167A
`ModelB`, 164A, 165A, 167A
`ModelC`, 164A, 166A, 167A
`Month`, 79A, 80A
`MP3Player`, 192A
`MsgInfo`, 208A
`MsgSender`, 208A
`MsgSender<CompanyZ>`, 209A
`NamedObject`, 35A, 36A
`NewHandlerHolder`, 243A
`NewHandlerSupport`, 245A
`output_iterator_tag`, 228A
`OutputFile`, 193A, 194A
`Penguin`, 151A, 152A, 153A
`Person`, 86A, 135A, 140A, 141A, 142A, 145A, 146A, 150A, 184A, 187A
`PersonInfo`, 195A, 197A
`PhoneNumber`, 27A, 184A
`PMImpl`, 131A
`Point`, 26A, 41A, 123A
`PrettyMenu`, 127A, 130A, 131A
`PriorityCustomer`, 58A
`random_access_iterator_tag`, 228A
`Rational`, 90A, 102A, 103A, 105A, 222A, 223A, 224A, 225A, 226A

`RealPerson`, 147A

`Rectangle`, 124A, 125A, 154A, 161A, 181A, 183A

`RectData`, 124A

`SellTransaction`, 49A

`Set`, 185A

`Shape`, 161A, 162A, 163A, 167A, 180A, 182A, 183A

`SmartPtr`, 218A, 219A, 220A

`SpecialString`, 42A

`SpecialWindow`, 119A, 120A, 121A, 122A

`SpeedDataCollection`, 96A

`Square`, 154A

`SquareMatrix`, 213A, 214A, 215A, 216A

`SquareMatrixBase`, 214A, 215A

`StandardNewDeleteForms`, 260A

`Student`, 86A, 150A, 187A

`TextBlock`, 20A, 23A, 24A

`TimeKeeper`, 40A, 41A

`Timer`, 188A

`Top`, 218A

`Transaction`, 48A, 50A, 51A

`Uncopyable`, 39A

`WaterClock`, 40A

`WebBrowser`, 98A, 100A, 101A

`Widget`, 4A, 5A, 44A, 52A, 53A, 54A, 56A, 107A, 108A, 109A, 118A, 189A, 199A, 201A, 242A, 245A, 246A, 257A, 258A, 261A

`Widget::WidgetTimer`, 189A

`WidgetImpl`, 106A, 108A

`Window`, 88A, 119A, 121A, 122A

`WindowWithScrollBars`, 88A

`WristWatch`, 40A

`X`, 242A

`Y`, 242A

`Year`, 79A

example functions/templates

`ABEntry::ABEntry`, 27A, 28A

`AccessLevels::getReadOnly`, 95A

`AccessLevels::getReadWrite`, 95A

`AccessLevels::setReadOnly`, 95A

`AccessLevels::setWriteOnly`, 95A

`advance`, 228A, 230A, 232A, 233A, 234A

`Airplane::defaultFly`, 165A

`Airplane::fly`, 164A, 165A, 166A, 167A

`askUserForDatabaseID`, 195A

`AWOV::AWOV`, 43A

`B::mf`, 178A

`Base::operator delete`, 255A

`Base::operator new`, 254A

`Bird::fly`, 151A

`BorrowableItem::checkOut`, 192A

`boundingBox`, 126A

`BuyTransaction::BuyTransaction`, 51A

`BuyTransaction::createLogString`, 51A

`calcHealth`, 174A

`callWithMax`, 16A

`changeFontSize`, 71A

`Circle::draw`, 181A

`clearAppointments`, 143A, 144A

`clearBrowser`, 98A

`CPerson::birthDate`, 198A

`CPerson::CPerson`, 198A

`CPerson::name`, 198A

`CPerson::valueDelimClose`, 198A

`CPerson::valueDelimOpen`, 198A

`createInvestment`, 62A, 70A, 81A, 82A, 83A

`CTextBlock::length`, 22A, 23A

`CTextBlock::operator[]`, 21A

`Customer::Customer`, 58A

`Customer::operator=`, 58A

`D::mf`, 178A

`Date::Date`, 79A

`Day::Day`, 79A

`daysHeld`, 69A

`DBConn::~DBConn`, 45A, 46A, 47A

`DBConn::close`, 47A

`defaultHealthCalc`, 172A, 173A

`Derived::Derived`, 138A, 206A

`Derived::mf1`, 160A

`Derived::mf4`, 157A

`Directory::Directory`, 31A, 32A

`doAdvance`, 231A

`doMultiply`, 226A

`doProcessing`, 200A, 202A

`doSomething`, 5A, 44A, 54A, 110A

`doSomeWork`, 118A

`eat`, 151A, 187A

`ElectronicGadget::checkOut`, 192A

`Empty::~Empty`, 34A

`Empty::Empty`, 34A

`Empty::operator=`, 34A

`encryptPassword`, 114A, 115A

`error`, 152A

`EvilBadGuy::EvilBadGuy`, 172A

`f`, 62A, 63A, 64A

`FlyingBird::fly`, 152A

`Font::~Font`, 71A

`Font::Font`, 71A

`Font::get`, 71A

`Font::operator FontHandle`, 71A

`GameCharacter::doHealthValue`, 170A

`GameCharacter::GameCharacter`, 172A, 174A, 176A

`GameCharacter::healthValue`, 169A, 170A, 172A, 174A, 176A

`GameLevel::health`, 174A

`getFont`, 70A

`hasAcceptableQuality`, 6A

`HealthCalcFunc::calc`, 176A

`HealthCalculator::operator()`, 174A

`lock`, 66A

`Lock::~Lock`, 66A

`Lock::Lock`, 66A, 68A

`logCall`, 57A

`LoggingMsgSender::sendClear`, 208A, 210A, 211A

`loseHealthQuickly`, 172A

`loseHealthSlowly`, 172A

`main`, 141A, 142A, 236A, 241A

`makeBigger`, 154A

`makePerson`, 195A

`max`, 135A

`ModelA::fly`, 165A, 167A

`ModelB::fly`, 165A, 167A

`ModelC::fly`, 166A, 167A

`Month::Dec`, 80A

`Month::Feb`, 80A

`Month::Jan`, 80A

`Month::Month`, 79A, 80A

`MsgSender::sendClear`, 208A

`MsgSender::sendSecret`, 208A

`MsgSender<CompanyZ>::sendSecret`, 209A

`NewHandlerHolder::~NewHandlerHolder`, 243A

`NewHandlerHolder::NewHandlerHolder`, 243A

`NewHandlerSupport::operator new`, 245A

`NewHandlerSupport::set_new_handler`, 245A

`numDigits`, 4A

`operator delete`, 255A

`operator new`, 249A, 252A

`operator*`, 91A, 92A, 94A, 105A, 222A, 224A, 225A, 226A

`operator==`, 93A

`outOfMem`, 240A

`Penguin::fly`, 152A

`Person::age`, 135A

`Person::create`, 146A, 147A

`Person::name`, 145A

`Person::Person`, 145A

`PersonInfo::theName`, 196A

`PersonInfo::valueDelimClose`, 196A

`PersonInfo::valueDelimOpen`, 196A

`PrettyMenu::changeBackground`, 127A, 128A, 130A, 131A

`print`, 20A

`print2nd`, 204A, 205A

`printNameAndDisplay`, 88A, 89A

`priority`, 75A

`PriorityCustomer::operator=`, 59A

`PriorityCustomer::PriorityCustomer`, 59A

`processWidget`, 75A

`RealPerson::~RealPerson`, 147A

`RealPerson::RealPerson`, 147A

`Rectangle::doDraw`, 183A

`Rectangle::draw`, 181A, 183A

`Rectangle::lowerRight`, 124A, 125A

`Rectangle::upperLeft`, 124A, 125A

`releaseFont`, 70A

`Set::insert`, 186A

`Set::member`, 186A

`Set::remove`, 186A

`Set::size`, 186A

`Shape::doDraw`, 183A

`Shape::draw`, 161A, 162A, 180A, 182A, 183A

`Shape::error`, 161A, 163A

`Shape::objectID`, 161A, 167A

`SmartPtr::get`, 220A

`SmartPtr::SmartPtr`, 220A

`someFunc`, 132A, 156A

`SpecialWindow::blink`, 122A

`SpecialWindow::onResize`, 119A, 120A

`SquareMatrix::invert`, 214A

`SquareMatrix::setDataPtr`, 215A

`SquareMatrix::SquareMatrix`, 215A, 216A

`StandardNewDeleteForms::operator delete`, 260A, 261A

`StandardNewDeleteForms::operator new`, 260A, 261A

`std::swap`, 109A

`std::swap<Widget>`, 107A, 108A

`study`, 151A, 187A

`swap`, 106A, 109A

`tempDir`, 32A

`TextBlock::operator[]`, 20A, 23A, 24A

`tfs`, 32A

`Timer::onTick`, 188A

`Transaction::init`, 50A

`Transaction::Transaction`, 49A, 50A, 51A

`Uncopyable::operator=`, 39A

`Uncopyable::Uncopyable`, 39A

`unlock`, 66A

`validateStudent`, 87A

`Widget::onTick`, 189A

`Widget::operator new`, 244A

`Widget::operator+=`, 53A

`Widget::operator=`, 53A, 54A, 55A, 56A, 107A

`Widget::set_new_handler`, 243A

`Widget::swap`, 108A

`Window::blink`, 122A
`Window::onResize`, 119A
`workWithIterator`, 206A, 207A
`Year::Year`, 79A
exception specifications, 85A
exceptions, 113A
`delete` and, 62A
  destructors and, 44–48A
  member `swap` and, 112A
  standard hierarchy for, 264A
  swallowing, 46A
  unused objects and, 114A
exception-safe code, 127–134A
  copy-and-`swap` and, 132A
  legacy code and, 133A
  pimpl idiom and, 131A
  side effects and, 132A
exception-safety guarantees, 128–129A
explicit calls to base class functions, 211A
`explicit` constructors, 5A, 85A, 104A
  generalized copy construction and, 219A
explicit inline request, 135A
explicit specification, of class names, 162A
explicit type conversions vs. implicit, 70–72A
expression templates, 237A
expressions, implicit interfaces and, 201A

F

factoring code, out of templates, 212–217A
factory function, 40A, 62A, 69A, 81A, 146A, 195A
federation, of languages, C++ as, 11–13A
`final` classes, in Java, 43A
`final` methods, in Java, 190A

fixed-size static buffers, problems of, 196A
forms of `new` and `delete`, 73–75A
FORTRAN, 42A
forward iterators, 227A
`forward_iterator_tag`, 228A
forwarding functions, 144A, 160A
friend functions, 38A, 85A, 105A, 135A, 173A, 223–225A
vs. member functions, 98–102A
friendship
    in real life, 105A
    without needing special access rights, 225A
`FUDGE_FACTOR`, 15A
function declarations, `const` in, 18A
function objects
    definition of, 6A
    higher-order programming utilities and, in Boost, 271A
functions
    convenience, 100A
    copying, 57A
    defining, 4A
    deliberately not defining, 38A
    factory, see factory function
    forwarding, 144A, 160A
    implicitly generated, 34–37A, 221A
    disallowing, 37–39A
    inline, declaring, 135A
    member
        templatized, 218–222A
        vs. non-member, 104–105A
    non-member
        templates and, 222–226A
        type conversions and, 102–105A, 222–226A
    non-member non-friend, vs member, 98–102A

  - non-virtual, meaning, 168A
  - return values, modifying, 21A
  - signatures, explicit interfaces and, 201A
  - static
    - ctors and dtors and, 52A
  - virtual, see virtual functions
- function-style casts, 116A

G

- generalized assignment, 220A
- generalized copy constructors, 219A
- generative programming, 237A
- generic programming support, in Boost, 271A
- `get`, smart pointers and, 70A
- goddess, see Urbano, Nancy L.
- `goto`, `delete` and, 62A
- Graph library, in Boost, 270A
- `grep`, casts and, 117A
- guarantees, exception safety, 128–129A

H

- Handle classes, 144–145A
- handles, 125A
  - dangling, 126A
  - encapsulation and, 124A
- `operator[]` and, 126A
  - returning, 123–126A
- has-a relationship, 184A
- hash tables, in TR1, 266A
- Hastings, Battle of, 150A
- head scratching, avoiding, 95A
- header files, see headers
- headers

  - for declarations vs. for definitions, 144A
  - inline functions and, 135A
  - namespaces and, 100A
  - of C++ standard library, 101A
  - templates and, 136A
  - usage, in this book, 3A
- hello world, template metaprogramming and, 235A
- hiding names, see name hiding
- higher-order programming and function object utilities, in Boost, 271A
- highlighting, in this book, 5A

I

- identity test, 55A
- `if...else` for types, 230A
- `#ifdef`, 17A
- `#ifndef`, 17A
- implementation-dependent behavior, warnings and, 263A
- implementations
  - decoupling from interfaces, 165A
  - default, danger of, 163–167A
  - inheritance of, 161–169A
  - of derived class constructors and destructors, 137A
  - of Interface classes, 147A
  - references, 89A
- `std::max`, 135A
- `std::swap`, 106A
- implicit inline request, 135A
- implicit interfaces, 199–203A
- implicit type conversions vs. explicit, 70–72A
- implicitly generated functions, 34–37A, 221A
  - disallowing, 37–39A
- `#include` directives, 17A
  - compilation dependencies and, 140A

incompatibilities, with built-in types, 80A
incorrect code and efficiency, 90A
infinite loop, in `operator new`, 253A
inheritance
    accidental, 165–166A
    combining with templates, 243–245A
    common features and, 164A
    intuition and, 151–155A
    mathematics and, 155A
    mixin-style, 244A
    name hiding and, 156–161A
    of implementation, 161–169A
    of interface, 161–169A
    of interface vs. implementation, 161–169A
`operator new` and, 253–254A
    penguins and birds and, 151–153A
    private, 187–192A
    protected, 151A
    public, 150–155A
    rectangles and squares and, 153–155A
    redefining non-virtual functions and, 178–180A
    scopes and, 156A
    sharing features and, 164A
inheritance, multiple, 192–198A
    ambiguity and, 192A
    combining public and private, 197A
    deadly diamond, 193A
inheritance, private, 214A
    combining with public, 197A
    eliminating, 189A
    for redefining virtual functions, 197A
    meaning, 187A
    vs. composition, 188A

inheritance, public
  combining with private, 197A
  is-a relationship and, 150–155A
  meaning of, 150A
  name hiding and, 159A
  virtual inheritance and, 194A
inheritance, virtual, 194A
`init` function, 60A
initialization, 4A, 26–27A
  assignment vs., 6A
  built-in types, 26–27A
  `const` members, 29A
  const static members, 14A
  default, unintended, 59A
  in-class, of static const integral members, 14A
  local static objects, 31A
  non-local static objects, 30A
  objects, 26–33A
  reference members, 29A
  static members, 242A
  virtual base classes and, 194A
  vs. assignment, 27–29A, 114A
  with vs. without arguments, 114A
initialization order
  class members, 29A
  importance of, 31A
  non-local statics, 29–33A
inline functions
  see also inlining
  address of, 136A
  as request to compiler, 135A
  debuggers and, 139A
  declaring, 135A

  - headers and, 135A
  - optimizing compilers and, 134A
  - recursion and, 136A
  - vs. `#define` 16–17A
  - vs. macros, efficiency and, 16A

inlining, 134–139A
  - constructors/destructors and, 137–138A
  - dynamic linking and, 139A
  - Handle classes and, 148A
  - inheritance and, 137–138A
  - Interface classes and, 148A
  - library design and, 138A
  - recompiling and, 139A
  - relinking and, 139A
  - suggested strategy for, 139A
  - templates and, 136A
  - time of, 135A
  - virtual functions and, 136A

input iterators, 227A

`input_iterator_tag`, 228A

`input_iterator_tag<Iter*>`, 230A

insomnia, 150A

instructions, reordering by compilers, 76A

integral types, 14A

Interface classes, 145–147A

interfaces
  - decoupling from implementations, 165A
  - definition of, 7A
  - design considerations, 78–86A
  - explicit, signatures and, 201A
  - implicit, 199–203A
    - expressions and, 201A
  - inheritance of, 161–169A

  new types and, 79–80A
  separating from implementations, 140A
  template parameters and, 199–203A
  undeclared, 85A
inter-language support, in Boost, 272A
internationalization, library support for, 264A
invalid array index, undefined behavior and, 7A
invariants
  NVI and, 171A
  over specialization, 168A
<iosfwd>, 144A
is-a relationship, 150–155A
is-implemented-in-terms-of, 184–186A, 187A
`istream_iterator`s, 227A
iterator categories, 227–228A
`iterator_category`, 229A
iterators as handles, 125A
`iterator`s, vs. `const_iterator`s, 18A

J

Java, 7A, 43A, 76A, 81A, 100A, 116A, 118A, 142A, 145A, 190A, 194A

K

Koenig lookup, 110A

L

Lambda library, in Boost, 271A
languages, other, compatibility with, 42A
late binding, 180A
layering, see composition
layouts, objects vs. arrays, 73A
leaks, exception-safe code and, 127A
legacy code, exception-safety and, 133A

lemur, ring-tailed, 196A
`lhs`, as parameter name, 8A
link-time errors, 39A, 44A
link-time inlining, 135A
`list`, 186A
local static objects
    definition of, 30A
    initialization of, 31A
locales, 264A
locks, RAII and, 66–68A
`logic_error` class, 113A
logically const member functions, 22–23A

M

maintenance
    common base classes and, 164A
`delete` and, 62A
managing resources, see resource management
math and numerics utilities, in Boost, 271A
mathematical functions, in TR1, 267A
mathematics, inheritance and, 155A
matrix operations, optimizing, 237A
`max`, `std`, implementation of, 135A
meaning
    of classes without virtual functions, 41A
    of composition, 184A
    of non-virtual functions, 168A
    of pass-by-value, 6A
    of private inheritance, 187A
    of public inheritance, 150A
    of pure virtual functions, 162A
    of references, 91A
    of simple virtual functions, 163A

measuring encapsulation, 99A
member data, see data members
member function templates, 218–222A
member functions
    bitwise const, 21–22A
    common design errors, 168–169A
`const`, 19–25A
    duplication and, 23–25A
    encapsulation and, 99A
    implicitly generated, 34–37A, 221A
        disallowing, 37–39A
    logically const, 22–23A
    private, 38A
    protected, 166A
    vs. non-member functions, 104–105A
    vs. non-member non-friends, 98–102A
member initialization
    for const static integral members, 14A
    lists, 28–29A
        vs. assignment, 28–29A
    order, 29A
memory allocation
    arrays and, 254–255A
    error handling for, 240–246A
memory leaks, `new` expressions and, 256A
memory management
    functions, replacing, 247–252A
    multithreading and, 239A, 253A
    utilities, in Boost, 272A
metaprogramming, see template metaprogramming
`mf`, as identifier, 9A
minimizing compilation dependencies, 140–148A, 190A
mixed-mode arithmetic, 103A, 104A, 222–226A

mixin-style inheritance, 244A
modeling is-implemented-in-terms of, 184–186A
modifying function return values, 21A
Monty Python, allusion to, 91A
  More Effective C++, 273A, 273–274A
  compared to *Effective C++*, 273A
  contents of, 273–274A
MPL library, in Boost, 270A, 271A
multiparadigm programming language, C++ as, 11A
multiple inheritance, see inheritance
multithreading
  memory management routines and, 239A, 253A
  non-`const` static objects and, 32A
  treatment in this book, 9A
`mutable`, 22–23A
mutexes, RAII and, 66–68A

N

name hiding
  inheritance and, 156–161A
  operators `new`/`delete` and, 259–261A
`using` declarations and, 159A
name lookup
`this->` and, 210A
`using` declarations and, 211A
name shadowing, see name hiding
names
  accessing in templatized bases, 207–212A
  available in both C and C++, 3A
  dependent, 204A
  hidden by derived classes, 263A
  nested, dependent, 204A
  non-dependent, 204A

namespaces, 110A
    headers and, 100A
    namespace pollution in a class, 166A
Nancy, see Urbano, Nancy L.
nested dependent names, 204A
nested dependent type names, `typename` and, 205A
`new` see also `operator new`
    expressions, memory leaks and, 256A
    forms of, 73–75A
`operator new` and, 73A
    relationship to constructors, 73A
    smart pointers and, 75–77A
new types, interface design and, 79–80A
new-handler, 240–247A
    definition of, 240A
    deinstalling, 241A
    identifying, 253A
new-handling functions, behavior of, 241A
new-style casts, 117A
`noncopyable` base class, in Boost, 39A
non-dependent names, 204A
non-local static objects, initialization of, 30A
non-member functions
    member functions vs., 104–105A
    templates and, 222–226A
    type conversions and, 102–105A, 222–226A
non-member non-friend functions, 98–102A
non-type parameters, 213A
non-virtual
    functions, 178–180A
        static binding of, 178A
    interface idiom, see NVI
nothrow guarantee, the, 129A

nothrow `new`, 246A
null pointer
    deleting, 255A
    dereferencing, 6A
`set_new_handler` and, 241A
NVI, 170–171A, 183A

O

object-oriented C++, as sublanguage of C++, 12A
object-oriented principles, encapsulation and, 99A
objects
    alignment of, 249–250A
    clustering, 251A
    compilation dependencies and, 143A
    copying all parts, 57–60A
    defining, 4A
    definitions, postponing, 113–116A
    handles to internals of, 123–126A
    initialization, with vs. without arguments, 114A
    layout vs. array layout, 73A
    multiple addresses for, 118A
    partial copies of, 58A
    placing in shared memory, 251A
    resource management and, 61–66A
    returning, vs. references, 90–94A
    size, pass-by-value and, 89A
    sizes, determining, 141A
    vs. variables, 3A
old-style casts, 117A
operations, reordering by compilers, 76A
`operator delete`, 84A
    see also `delete`
    behavior of, 255A

  - efficiency of, 248A
  - name hiding and, 259–261A
  - non-member, pseudocode for, 255A
  - placement, 256–261A
  - replacing, 247–252A
  - standard forms of, 260A
  - virtual destructors and, 255A
- `operator delete[]`, 84A, 255A
- `operator new`, 84A
  - see also `new`
  - arrays and, 254–255A
- `bad_alloc` and, 246A, 252A
  - behavior of, 252–255A
  - efficiency of, 248A
  - infinite loop within, 253A
  - inheritance and, 253–254A
  - member, and “wrongly sized” requests, 254A
  - name hiding and, 259–261A
  - new-handling functions and, 241A
  - non-member, pseudocode for, 252A
  - out-of-memory conditions and, 240–241A, 252–253A
  - placement, 256–261A
  - replacing, 247–252A
  - returning, 0 and, 246A
  - standard forms of, 260A
- `std::bad_alloc` and, 246A, 252A
- `operator new[]`, 84A, 254–255A
- `operator()` (function call operator), 6A
- `operator=const` members and, 36–37A
  - default implementation, 35A
  - implicit generation, 34A
  - reference members and, 36–37A
  - return value of, 52–53A

  - self-assignment and, 53–57A
  - when not implicitly generated, 36–37A
- `operator[]`, 126A
  - overloading on `const` 19–20A
  - return type of, 21A
- optimization
  - by compilers, 94A
  - during compilation, 134A
    - inline functions and, 134A
- order
  - initialization of non-local statics, 29–33A
  - member initialization, 29A
- `ostream_iterator`s, 227A
- other languages, compatibility with, 42A
- output iterators, 227A
- `output_iterator_tag`, 228A
- overloading
  - as `if`...`else` for types, 230A
  - on `const` 19–20A
- `std::swap`, 109A
- overrides of virtuals, preventing, 189A
- ownership transfer, 68A

P

- parameters
  - see also pass-by-value, pass-by-reference
  - default, 180–183A
  - evaluation order, 76A
  - non-type, for templates, 213A
  - type conversions and, see type conversions
- Pareto Principle, see, 80–20 rule
- parsing problems, nested dependent names and, 204A
- partial copies, 58A

partial specialization
  function templates, 109A
`std::swap`, 108A
parts, of objects, copying all, 57–60A
pass-by-reference, efficiency and, 87A
pass-by-reference-to-`const`, vs pass-by-value, 86–90A
pass-by-value
  copy constructor and, 6A
  efficiency of, 86–87A
  meaning of, 6A
  object size and, 89A
  vs. pass-by-reference-to-`const`, 86–90A
patterns
  see design patterns
penguins and birds, 151–153A
performance, see efficiency
Persephone, 36A
pessimization, 93A
physical constness, see `const`, bitwise
pimpl idiom
  definition of, 106A
  exception-safe code and, 131A
placement `delete`, see `operator delete`
placement `new`, see `operator new`
Plato, 87A
pointer arithmetic and undefined behavior, 119A
pointers
  see also smart pointers
  as handles, 125A
  bitwise `const` member functions and, 21A
  compilation dependencies and, 143A
`const`, 17A
    in headers, 14A

- null, dereferencing, 6A
- template parameters and, 217A
- to single vs. multiple objects, and `delete`, 73A

polymorphic base classes, destructors and, 40–44A

polymorphism, 199–201A
- compile-time, 201A
- runtime, 200A

Pool library, in Boost, 250A, 251A

postponing variable definitions, 113–116A

preconditions, NVI and, 171A

pregnancy, exception-safe code and, 133A

private data members, why, 94–98A

private inheritance, see inheritance

private member functions, 38A

private virtual functions, 171A

properties, 97A

protected
- data members, 97A
- inheritance, see inheritance
- member functions, 166A
- members, encapsulation of, 97A

public inheritance, see inheritance

pun, really bad, 152A

pure virtual destructors
- defining, 43A
- implementing, 43A

pure virtual functions, 43A
- defining, 162A, 166–167A
- meaning, 162A

R

RAII, 63A, 70A, 243A
- classes, 72A

- copying behavior and, 66–69A
  - encapsulation and, 72A
  - mutexes and, 66–68A
- random access iterators, 227A
- random number generation, in TR1, 267A
- `random_access_iterator_tag`, 228A
- RCSP, see smart pointers
- reading uninitialized values, 26A
- rectangles and squares, 153–155A
- recursive functions, inlining and, 136A
- redefining inherited non-virtual functions, 178–180A
- references
  - as handles, 125A
  - compilation dependencies and, 143A
  - functions returning, 31A
  - implementation, 89A
  - meaning, 91A
  - members, initialization of, 29A
  - returning, 90–94A
  - to static object, as function return value, 92–94A
- register usage, objects and, 89A
- regular expressions, in TR1, 266A
- `reinterpret_cast`, 117A, 249A
  - see also casting
- relationships
  - has-a, 184A
  - is-a, 150–155A
  - is-implemented-in-terms-of, 184–186A, 187A
- reordering operations, by compilers, 76A
- replacing definitions with declarations, 143A
- replacing `new`/`delete` 247–252A
- replication, see duplication
- Resource Acquisition Is Initialization, see RAII

resource leaks, exception-safe code and, 127A
resource management
    see also RAII
    copying behavior and, 66–69A
    objects and, 61–66A
    raw resource access and, 69–73A
resources, managing objects and, 69–73A
return by reference, 90–94A
return types
`const`, 18A
    objects vs. references, 90–94A
    of `operator[]`, 21A
return value of `operator=`, 52–53A
returning handles, 123–126A
reuse, see code reuse
revenge, compilers taking, 58A
`rhs`, as parameter name, 8A
rule of, 80–20, 139A, 168A
runtime
    errors, 152A
    inlining, 135A
    polymorphism, 200A

S

Satch, 36A
`scoped_array`, 65A, 216A, 272A
scopes, inheritance and, 156A
`sealed` classes, in C#, 43A
`sealed` methods, in C#, 190A
second edition, see, 2nd edition
self-assignment, `operator=` and, 53–57A
`set`, 185A
`set_new_handler`
    class-specific, implementing, 243–245A

- using, 240–246A

`set_unexpected` function, 129A
shadowing, names, see name shadowing
Shakespeare, William, 156A
shared memory, placing objects in, 251A
`shared_array`, 65A
`shared_ptr` implementation in Boost, costs, 83A
sharing code, see duplication, avoiding
sharing common features, 164A
side effects, exception safety and, 132A
signatures
- definition of, 3A
- explicit interfaces and, 201A

simple virtual functions, meaning of, 163A
Singleton pattern, 31A
`size_t`, 3A
`sizeof`, 253A, 254A
- empty classes and, 190A
- freestanding classes and, 254A

sizes
- of freestanding classes, 254A
- of objects, 141A

sleeping pills, 150A
`slist`, 227A
Smalltalk, 142A
smart pointers, 63A, 64A, 70A, 81A, 121A, 146A, 237A
- see also `std::auto_ptr` and `tr1::shared_ptrget` and, 70A
- in Boost, 65A, 272A
- in TR1, 265A

`new`ed objects and, 75–77A
- type conversions and, 218–220A

Socrates, 87A
Some Must Watch While Some Must Sleep, 150A

specialization
  invariants over, 168A
  partial, of `std::swap`, 108A
  total, of `std::swap`, 107A, 108A
specification, see interfaces
squares and rectangles, 153–155A
standard exception hierarchy, 264A
standard forms of `operator new/delete`, 260A
standard library, see C++ standard library, C standard library
standard template library, see STL
statements using `new` , smart pointers and, 75–77A
static
  binding
    of default parameters, 182A
    of non-virtual functions, 178A
    objects, returning references to, 92–94A
  type, definition of, 180A
static functions, ctors and dtors and, 52A
static members
`const` member functions and, 21A
  definition, 242A
  initialization, 242A
static objects
  definition of, 30A
  multithreading and, 32A
`static_cast`, 25A, 82A, 117A, 119A, 249A
  see also casting
`std` namespace, specializing templates in, 107A
`std::auto_ptr`, 63–65A, 70A
  conversion to `tr1::shared_ptr` and, 220A
`delete []` and, 65A
  pass by `const` and, 220A
`std::auto_ptr`, deleter support and, 68A

`std::char_traits`, 232A
`std::iterator_trait`s, pointers and, 230A
`std::list`, 186A
`std::max`, implementation of, 135A
`std::numeric_limits`, 232A
`std::set`, 185A
`std::size_t`, 3A
`std::swap` see also `swap`
  implementation of, 106A
  overloading, 109A
  partial specialization of, 108A
  total specialization of, 107A, 108A
`std::tr1`, see TR1
stepping through functions, inlining and, 139A
STL
  allocators, 240A
  as sublanguage of C++, 12A
  containers, `swap` and, 108A
  definition of, 6A
  iterator categories in, 227–228A
Strategy pattern, 171–177A
string and text utilities, in Boost, 271A
strong guarantee, the, 128A
swallowing exceptions, 46A
`swap`, 106–112A
  see also `std::swap`
  calling, 110A
  exceptions and, 112A
  STL containers and, 108A
  when to write, 111A
symbols, available in both C and C++, 3A

T

template C++, as sublanguage of C++, 12A
template metaprogramming, 233–238A
  efficiency and, 233A
  hello world in, 235A
  pattern implementations and, 237A
  support in Boost, 271A
  support in TR1, 267A
Template Method pattern, 170A
templates
  code bloat, avoiding in, 212–217A
  combining with inheritance, 243–245A
  defining, 4A
  errors, when detected, 212A
  expression, 237A
  headers and, 136A
  in `std`, specializing, 107A
  inlining and, 136A
  instantiation of, 222A
  member functions, 218–222A
  names in base classes and, 207–212A
  non-type parameters, 213A
  parameters, omitting, 224A
  pointer type parameters and, 217A
  shorthand for, 224A
  specializations, 229A, 235A
    partial, 109A, 230A
    total, 107A, 209A
  type conversions and, 222–226A
  type deduction for, 223A
temporary objects, eliminated by compilers, 94A
terminology, used in this book, 3–8A
testing and correctness, Boost support for, 272A
text and string utilities, in Boost, 271A

third edition, see, 3rd edition
`this->`, to force base class lookup, 210A
threading, see multithreading
TMP, see template metaprogramming
total class template specialization, 209A
total specialization of `std::swap`, 107A, 108A
total template specializations, 107A
TR1, 9A, 264–267A
`array` component, 267A
`bind` component, 266A
    Boost and, 9–10A, 268A, 269A
`boost` as synonym for `std::tr1`, 268A
    C99 compatibility component, 267A
`function` component, 265A
    hash tables component, 266A
    math functions component, 267A
`mem_fn` component, 267A
    random numbers component, 267A
`reference_wrapper` component, 267A
    regular expression component, 266A
`result_of` component, 267A
    smart pointers component, 265A
    support for TMP, 267A
    tuples component, 266A
    type traits component, 267A
    URL for information on, 268A
`tr1::array`, 267A
`tr1::bind`, 175A, 266A
`tr1::function`, 173–175A, 265A
`tr1::mem_fn`, 267A
`tr1::reference_wrapper`, 267A
`tr1::result_of`, 267A
`tr1::shared_ptr`, 53A, 64–65A, 70A, 75–77A

  - construction from other smart pointers and, 220A
  - cross-DLL problem and, 82A

`delete []` and, 65A

  - deleter support in, 68A, 81–83A
  - member template ctors in, 220–221A

`tr1::tuple`, 266A

`tr1::unordered_map`, 43A, 266A

`tr1::unordered_multimap`, 266A

`tr1::unordered_multiset`, 266A

`tr1::unordered_set`, 266A

`tr1::weak_ptr`, 265A

traits classes, 226–232A

transfer, ownership, 68A

translation unit, definition of, 30A

tuples, in TR1, 266A

type conversions, 85A, 104A

`explicit` ctors and, 5A

  - implicit, 104A
  - implicit vs. explicit, 70–72A
  - non-member functions and, 102–105A, 222–226A
  - private inheritance and, 187A
  - smart pointers and, 218–220A
  - templates and, 222–226A

type deduction, for templates, 223A

type design, 78–86A

type traits, in TR1, 267A

`typedef`, `typename` and, 206–207A

typedefs, `new`/`delete` and, 75A

`typeid`, 50A, 230A, 234A, 235A

typelists, 271A

`typename` 203–207A

  - compiler variations and, 207A

`typedef` and, 206–207A

  - vs. `class`, 203A
- types
  - built-in, initialization, 26–27A
  - compatible, accepting all, 218–222A
- `if`...`else` for, 230A
  - integral, definition of, 14A
  - traits classes and, 226–232A

U

- undeclared interface, 85A
- undefined behavior
- `advance` and, 231A
  - array deletion and, 73A
  - casting + pointer arithmetic and, 119A
  - definition of, 6A
  - destroyed objects and, 91A
  - exceptions and, 45A
  - initialization order and, 30A
  - invalid array index and, 7A
  - multiple `delete`s and, 63A, 247A
  - null pointers and, 6A
  - object deletion and, 41A, 43A, 74A
  - uninitialized values and, 26A
- undefined values of members before construction and after destruction, 50A
- `unexpected` function, 129A
- uninitialized
  - data members, virtual functions and, 49A
  - values, reading, 26A
- unnecessary objects, avoiding, 115A
- unused objects
  - cost of, 113A
  - exceptions and, 114A
- URLs

    Boost, 10A, 269A, 272A
Effective C++ TR1 Info. Page, 268A
usage statistics, memory management and, 248A
`using` declarations
    name hiding and, 159A
    name lookup and, 211A

V

`valarray`, 264A
value, pass by, see pass-by-value
variable, vs. object, 3A
variables definitions, postponing, 113–116A
`vector` template, 75A
virtual base classes, 193A
virtual constructors, 146A, 147A
virtual destructors
`operator delete` and, 255A
    polymorphic base classes and, 40–44A
virtual functions
    alternatives to, 169–177A
    ctors/dtors and, 48–52A
    default implementations and, 163–167A
    default parameters and, 180–183A
    dynamic binding of, 179A
    efficiency and, 168A
    explict base class qualification and, 211A
    implementation, 42A
    inlining and, 136A
    language interoperability and, 42A
    meaning of none in class, 41A
    preventing overrides, 189A
    private, 171A
    pure, see pure virtual functions

  simple, meaning of, 163A
  uninitialized data members and, 49A
virtual inheritance, see inheritance
virtual table, 42A
virtual table pointer, 42A
`vptr`, 42A
`vtbl`, 42A

W

warnings, from compiler, 262–263A
  calls to virtuals and, 50A
  inlining and, 136A
  partial copies and, 58A
web sites, see URLs
`Widget` class, as used in this book, 8A
Wizard of Oz, allusion to, 154A

X

XP, allusion to, 225A
XYZ Airlines, 163A

Z

Zabluda, Oleg xviii
Zolman, Leor xviii, xix

Footnotes

Chapter 1

[†] Some people find it helpful to read pointer declarations right to left, e.g., to read `const char * const p` **as** “p is a constant pointer to constant chars.“

[†] Actually, it‘s only *part* of a Singleton implementation. An essential part of Singleton I ignore in this Item is preventing the creation of multiple objects of a particular type.

Chapter 2

[†] Probably. C++ implementations are permitted to change the value of a deleted pointer (e.g., to null or some other special bit pattern), but I am unaware of any that do.

Chapter 5

[†] For information on the unexpected function, consult your favorite search engine or comprehensive C++ text. (You'll probably have better luck searching for set_unexpected, the function that specifies the unexpected function.)

Chapter 9

[1] As I write this in early 2005, the document has not been finalized, and its URL is subject to change. I therefore suggest you consult the *Effective C++* TR1 Information Page, http://aristeia.com/EC3E/TR1_info.html. That URL will remain stable.

More Effective C++

35 New Ways to Improve Your Programs and Designs

Scott Meyers

ADDISON–WESLEY

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearsoned.com

Library of Congress Cataloging-in-Publication Data

Meyers, Scott (Scott Douglas)

More effective C++: 35 new ways to improve your programs and designs / Scott Meyers.
p. cm.—(Addison-Wesley professional computing series)

Includes bibliographical references and index.

ISBN 0-201-63371-X (paperback: alk. paper)

1. C++ (Computer program language) I. Series.

QA76.73.C153M495 1996
005.13'3—dc20 95–47354

CIP

Copyright © 1996 by Addison-Wesley

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

ISBN-13: 978-0-201-63371-9
ISBN-10: 0-201-63371-X

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.

Twenty-fourth printing April 2007

For Clancy,
my favorite enemy within.

Praise for *More Effective C++: 35 New Ways to Improve Your Programs and Designs*

"This is an enlightening book on many aspects of C++: both the regions of the language you seldom visit, and the familiar ones you THOUGHT you understood. Only by understanding deeply how the C++ compiler interprets your code can you hope to write robust software using this language. This book is an invaluable resource for gaining that level of understanding. After reading this book, I feel like I've been through a code review with a master C++ programmer, and picked up many of his most valuable insights."

—Fred Wild, Vice President of Technology, Advantage Software Technologies

"This book includes a great collection of important techniques for writing programs that use C++ well. It explains how to design and implement the ideas, and what hidden pitfalls lurk in some obvious alternative designs. It also includes clear explanations of features recently added to C++. Anyone who wants to use these new features will want a copy of this book close at hand for ready reference."

—Christopher J. Van Wyk, Professor, Mathematics and Computer Science, Drew University

"Industrial strength C++ at its best. The perfect companion to those who have read Effective C++.*"*

—Eric Nagler, C++ Instructor and Author, University of California Santa Cruz Extension

"More Effective C++ *is a thorough and valuable follow-up to Scott's first book,* Effective C++. *I believe that every professional C++ developer should read and commit to memory the tips in both* Effective C++ *and* More Effective C++. *I've found that the tips cover poorly understood, yet important and sometimes arcane facets of the language. I strongly recommend this book, along with his first, to developers, testers, and managers ... everyone can benefit from his expert knowledge and excellent presentation."*

—Steve Burkett, Software Consultant

Acknowledgments

A great number of people helped bring this book into existence. Some contributed ideas for technical topics, some helped with the process of producing the book, and some just made life more fun while I was working on it.

When the number of contributors to a book is large, it is not uncommon to dispense with individual acknowledgments in favor of a generic "Contributors to this book are too numerous to mention." I prefer to follow the expansive lead of John L. Hennessy and David A. Patterson in *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann, first edition 1990). In addition to motivating the comprehensive acknowledgments that follow, their book provides hard data for the 90-10 rule, which I refer to in Item 16.

The Items

With the exception of direct quotations, all the words in this book are mine. However, many of the ideas I discuss came from others. I have done my best to keep track of who contributed what, but I know I have included information from sources I now fail to recall, foremost among them many posters to the Usenet newsgroups `comp.lang.c++` and `comp.std.c++`.

Many ideas in the C++ community have been developed independently by many people. In what follows, I note only where *I* was exposed to particular ideas, not necessarily where those ideas originated.

Brian Kernighan suggested the use of macros to approximate the syntax of the new C++ casting operators I describe in Item 2.

In Item 3, my warning about deleting an array of derived class objects through a base class pointer is based on material in Dan Saks' "Gotchas" talk, which he's given at several conferences and trade shows.

In Item 5, the proxy class technique for preventing unwanted application of single-argument constructors is based on material in Andrew Koenig's column in the January 1994 *C++ Report*.

James Kanze made a posting to `comp.lang.c++` on implementing postfix increment and decrement operators via the corresponding prefix functions; I use his technique in Item 6.

David Cok, writing me about material I covered in *Effective C++*, brought to my attention the distinction between `operator new` and the `new` operator that is the crux of Item 8. Even after reading his letter, I didn't really understand the distinction, but without his initial prodding, I probably *still* wouldn't.

The notion of using destructors to prevent resource leaks (used in Item 9) comes from section 15.3 of Margaret A. Ellis' and Bjarne Stroustrup's *The Annotated C++ Reference Manual* (see page 285B). There the technique is called *resource acquisition is initialization*. Tom Cargill suggested I shift the focus of the approach from resource acquisition to resource release.

Some of my discussion in Item 11 was inspired by material in Chapter 4 of *Taligent's Guide to Designing Programs* (Addison-Wesley, 1994).

My description of over-eager memory allocation for the `DynArray` class in Item 18 is based on Tom Cargill's article, "A Dynamic vector is harder than it looks," in the June 1992 *C++ Report*. A more sophisticated design for a dynamic array class can be found in Cargill's follow-up column in the January 1994 *C++ Report*.

Item 21 was inspired by Brian Kernighan's paper, "An AWK to C++ Translator," at the 1991 USENIX C++ Conference. His use of overloaded operators (sixty-seven of them!) to handle mixed-type arithmetic operations, though designed to solve a problem unrelated to the one I explore in Item 21, led me to consider multiple overloadings as a solution to the problem of temporary creation.

In Item 26, my design of a template class for counting objects is based on a posting to `comp.lang.c++` by Jamshid Afshar.

The idea of a mixin class to keep track of pointers from `operator new` (see Item 27) is based on a suggestion by Don Box. Steve Clamage made the idea practical by explaining how `dynamic_cast` can be used to find the beginning of memory for an object.

The discussion of smart pointers in Item 28 is based in part on Steven Buroff's and Rob Murray's *C++ Oracle* column in the October 1993 *C++ Report*; on Daniel R. Edelson's classic paper, "Smart Pointers: They're Smart, but They're Not Pointers," in the proceedings of the 1992 USENIX C++ Conference; on section 15.9.1 of Bjarne Stroustrup's *The Design and Evolution of C++* (see page 285B); on Gregory Colvin's "C++ Memory Management" class notes from C/C++ Solutions '95; and on Cay Horstmann's column in the March-April 1993 issue of the *C++ Report*. I developed some of the material myself, though. Really.

In Item 29, the use of a base class to store reference counts and of smart pointers to manipulate those counts is based on Rob Murray's discussions of the same topics in sections 6.3.2 and 7.4.2, respectively, of his *C++ Strategies and Tactics* (see page 286B). The design for adding reference counting to existing classes follows that presented by Cay Horstmann in his March-April 1993 column in the *C++ Report*.

In Item 30, my discussion of lvalue contexts is based on comments in Dan Saks' column in the *C User's Journal* (now the *C/C++ Users Journal*) of January 1993. The observation that non-proxy member functions are unavailable when called through proxies comes from an unpublished paper by Cay Horstmann.

The use of runtime type information to build vtbl-like arrays of function pointers (in Item 31) is based on ideas put forward by Bjarne Stroustrup in postings to `comp.lang.c++` and in section 13.8.1 of his *The Design and Evolution of C++* (see page 285B).

The material in Item 33 is based on several of my *C++ Report* columns in 1994 and 1995. Those columns, in turn, included comments I received from Klaus Kreft about how to use `dynamic_cast` to implement a virtual `operator=` that detects arguments of the wrong type.

Much of the material in Item 34 was motivated by Steve Clamage's article, "Linking C++ with other languages," in the May 1992 *C++ Report*. In that same Item, my treatment of the problems caused by functions like `strdup` was motivated by an anonymous reviewer.

The Book

Reviewing draft copies of a book is hard — and vitally important — work. I am grateful that so many people were willing to invest their time and energy on my behalf. I am especially grateful to Jill Huchital, Tim Johnson, Brian

Kernighan, Eric Nagler, and Chris Van Wyk, as they read the book (or large portions of it) more than once. In addition to these gluttons for punishment, complete drafts of the manuscript were read by Katrina Avery, Don Box, Steve Burkett, Tom Cargill, Tony Davis, Carolyn Duby, Bruce Eckel, Read Fleming, Cay Horstmann, James Kanze, Russ Paielli, Steve Rosenthal, Robin Rowe, Dan Saks, Chris Sells, Webb Stacy, Dave Swift, Steve Vinoski, and Fred Wild. Partial drafts were reviewed by Bob Beauchaine, Gerd Hoeren, Jeff Jackson, and Nancy L. Urbano. Each of these reviewers made comments that greatly improved the accuracy, utility, and presentation of the material you find here.

Once the book came out, I received corrections and suggestions from many people: Luis Kida, John Potter, Tim Uttormark, Mike Fulkerson, Dan Saks, Wolfgang Glunz, Clovis Tondo, Michael Loftus, Liz Hanks, Wil Evers, Stefan Kuhlins, Jim McCracken, Alan Duchan, John Jacobsma, Ramesh Nagabushnam, Ed Willink, Kirk Swenson, Jack Reeves, Doug Schmidt, Tim Buchowski, Paul Chisholm, Andrew Klein, Eric Nagler, Jeffrey Smith, Sam Bent, Oleg Shteynbuk, Anton Doblmaier, Ulf Michaelis, Sekhar Muddana, Michael Baker, Yechiel Kimchi, David Papurt, Ian Haggard, Robert Schwartz, David Halpin, Graham Mark, David Barrett, Damian Kanarek, Ron Coutts, Lance Whitesel, Jon Lachelt, Cheryl Ferguson, Munir Mahmood, Klaus-Georg Adams, David Goh, Chris Morley, Rainer Baumschlager, Christopher Tavares, Brian Kernighan, Charles Green, Mark Rodgers, Bobby Schmidt, Sivaramakrishnan J., Eric Anderson, Phil Brabbin, Feliks Kluzniak, Evan McLean, Kurt Miller, Niels Dekker, Balog Pal, Dean Stanton, William Mattison, Chulsu Park, Pankaj Datta, John Newell, Ani Taggu, Christopher Creutzi, Chris Wineinger, Alexander Bogdanchikov, Michael Tegtmeyer, Aharon Robbins, Davide Gennaro, Adrian Spermezan, Matthias Hofmann, Chang Chen, John Wismar, Mark Symonds, Thomas Kim, Ita Ryan, Rice Yeh, and Colas Schretter. Their suggestions allowed me to improve *More Effective C++* in updated printings (such as this one), and I greatly appreciate their help.

During preparation of this book, I faced many questions about the emerging ISO/ANSI standard for C++, and I am grateful to Steve Clamage and Dan Saks for taking the time to respond to my incessant email queries.

John Max Skaller and Steve Rumsby conspired to get me the HTML for the draft ANSI C++ standard before it was widely available. Vivian Neou pointed me to the Netscape WWW browser as a stand-alone HTML viewer under (16 bit) Microsoft Windows, and I am deeply grateful to the folks at Netscape Communications for making their fine viewer freely available on such a pathetic excuse for an operating system.

Bryan Hobbs and Hachemi Zenad generously arranged to get me a copy of the internal engineering version of the MetaWare C++ compiler so I could check the code in this book using the latest features of the language. Cay Horstmann helped me get the compiler up and running in the very foreign world of DOS and DOS extenders. Borland provided a beta copy of their most advanced compiler, and Eric Nagler and Chris Sells provided invaluable help in testing code for me on compilers to which I had no access.

Without the staff at the Corporate and Professional Publishing Division of Addison-Wesley, there would be no book, and I am indebted to Kim Dawley, Lana Langlois, Simone Payment, Marty Rabinowitz, Pradeepa Siva, John Wait, and the rest of the staff for their encouragement, patience, and help with the production of this work.

Chris Guzikowski helped draft the back cover copy for this book, and Tim Johnson stole time from his research on low-temperature physics to critique later versions of that text.

Tom Cargill graciously agreed to make his *C++ Report* article on exceptions (see page 287B) available at the Addison-Wesley Internet site.

The People

Kathy Reed was responsible for my introduction to programming; surely she didn't deserve to have to put up with a kid like me. Donald French had faith in my ability to develop and present C++ teaching materials when I had no track record. He also introduced me to John Wait, my editor at Addison-Wesley, an act for which I will always be grateful. The triumvirate at Beaver Ridge — Jayni Besaw, Lorri Fields, and Beth McKee — provided untold entertainment on my breaks as I worked on the book.

My wife, Nancy L. Urbano, put up with me and put up with me and put up with me as I worked on the book, continued to work on the book, and kept working on the book. How many times did she hear me say we'd do something after the book was done? Now the book is done, and we will do those things. She amazes me. I love her.

Finally, I must acknowledge our puppy, Persephone, whose existence changed our world forever. Without her, this book would have been finished both sooner and with less sleep deprivation, but also with substantially less comic relief.

Introduction

These are heady days for C++ programmers. Commercially available less than a decade, C++ has nevertheless emerged as the language of choice for systems programming on nearly all major computing platforms. Companies and individuals with challenging programming problems increasingly embrace the language, and the question faced by those who do not use C++ is often *when* they will start, not *if*. Standardization of C++ is complete, and the breadth and scope of the accompanying library — which both dwarfs and subsumes that of C — makes it possible to write rich, complex programs without sacrificing portability or implementing common algorithms and data structures from scratch. C++ compilers continue to proliferate, the features they offer continue to expand, and the quality of the code they generate continues to improve. Tools and environments for C++ development grow ever more abundant, powerful, and robust. Commercial libraries all but obviate the need to write code in many application areas.

As the language has matured and our experience with it has increased, our needs for information about it have changed. In 1990, people wanted to know *what* C++ was. By 1992, they wanted to know *how* to make it work. Now C++ programmers ask higher-level questions: How can I design my software so it will adapt to future demands? How can I improve the efficiency of my code without compromising its correctness or making it harder to use? How can I implement sophisticated functionality not directly supported by the language?

In this book, I answer these questions and many others like them.

This book shows how to design and implement C++ software that is *more effective*: more likely to behave correctly; more robust in the face of exceptions; more efficient; more portable; makes better use of language features; adapts to change more gracefully; works better in a mixed-language environment; is easier to use correctly; is harder to use incorrectly. In short, software that's just *better*.

The material in this book is divided into 35 Items. Each Item summarizes accumulated wisdom of the C++ programming community on a particular topic. Most Items take the form of guidelines, and the explanation accompanying each guideline describes why the guideline exists, what happens if you fail to follow it, and under what conditions it may make sense to violate the guideline anyway.

Items fall into several categories. Some concern particular language features, especially newer features with which you may have little experience. For example, Items 9 through 15 are devoted to exceptions. Other Items explain how to combine the features of the language to achieve higher-level goals. Items 25 through 31, for instance, describe how to constrain the number or placement of objects, how to create functions that act "virtual" on the type of more than one object, how to create "smart pointers," and more. Still other Items address broader topics; Items 16 through 24 focus on efficiency. No matter what the topic of a particular Item, each takes a no-nonsense approach to the subject. In *More Effective C++*, you learn how to *use* C++ more effectively. The descriptions of language features that make up the bulk of most C++ texts are in this book mere background information.

An implication of this approach is that you should be familiar with C++ before reading this book. I take for granted that you understand classes, protection levels, virtual and nonvirtual functions, etc., and I assume you are acquainted with the concepts behind templates and exceptions. At the same time, I don't expect you to be a language expert, so when poking into lesser-known corners of C++, I always explain what's going on.

The C++ in *More Effective C++*

The C++ I describe in this book is the language specified by the 1998 International Standard for C++. This means I may use a few features your compilers don't yet support. Don't worry. The only "new" feature I assume you have is templates, and templates are now almost universally available. I use exceptions, too, but that use is largely confined to Items 9 through 15, which are specifically devoted to exceptions. If you don't have access to a compiler offering exceptions, that's okay. It won't affect your ability to take advantage of the material in the other parts of the book. Furthermore, you should read Items 9 through 15 even if you don't have support for exceptions, because those items examine issues you need to understand in any case.

I recognize that just because the standardization committee blesses a feature or endorses a practice, there's no guarantee that the feature is present in current compilers or the practice is applicable to existing environments. When faced with a discrepancy between theory (what the committee says) and practice (what actually works), I discuss both, though my bias is toward things that work. Because I discuss both, this book will aid you as your compilers approach conformance with the standard. It will show you how to use existing constructs to approximate language features your compilers don't yet support, and it will guide you when you decide to transform workarounds into newly-supported features.

Notice that I refer to your *compilers* — plural. Different compilers implement varying approximations to the standard, so I encourage you to develop your code under at least two compilers. Doing so will help you avoid inadvertent dependence on one vendor's proprietary language extension or its misinterpretation of the standard. It will also help keep you away from the bleeding edge of compiler technology, e.g., from new features supported by only one vendor. Such features are often poorly implemented (buggy or slow — frequently both), and upon their introduction, the C++ community lacks experience to advise you in their proper use. Blazing trails can be exciting, but when your goal is producing reliable code, it's often best to let others test the waters before jumping in.

There are two constructs you'll see in this book that may not be familiar to you. Both are relatively recent language extensions. Some compilers support them, but if your compilers don't, you can easily approximate them with features you do have.

The first construct is the `bool` type, which has as its values the keywords `true` and `false`. If your compilers haven't implemented `bool`, there are two ways to approximate it. One is to use a global enum:

```
enum bool { false, true };
```

This allows you to overload functions on the basis of whether they take a `bool` or an `int`, but it has the disadvantage that the built-in comparison operators (i.e., `==`, `<`, `>=`, etc.) still return `int`s. As a result, code like the following will not behave the way it's supposed to:

Click here to view code image

```
void f(int);
void f(bool);
```

```
int x, y;
...
f( x < y );                           // calls f(int), but it
                                      // should call f(bool)
```

The enum approximation may thus lead to code whose behavior changes when you submit it to a compiler that truly supports `bool`.

An alternative is to use a typedef for `bool` and constant objects for `true` and `false`:

```
typedef int bool;

const bool false = 0;
const bool true = 1;
```

This is compatible with the traditional semantics of C and C++, and the behavior of programs using this approximation won't change when they're ported to `bool`-supporting compilers. The drawback is that you can't differentiate between `bool` and `int` when overloading functions. Both approximations are reasonable. Choose the one that best fits your circumstances.

The second new construct is really four constructs, the casting forms `static_cast`, `const_cast`, `dynamic_cast`, and `reinterpret_cast`. If you're not familiar with these casts, you'll want to turn to Item 2 and read all about them. Not only do they do more than the C-style casts they replace, they do it better. I use these new casting forms whenever I need to perform a cast in this book.

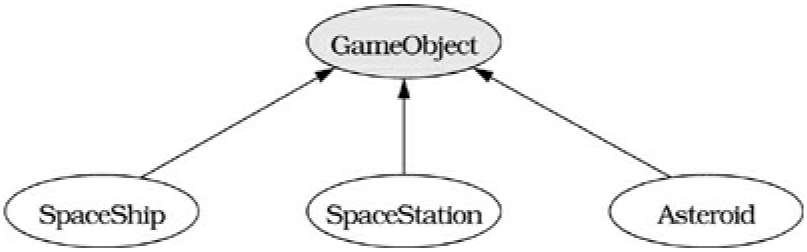
There is more to C++ than the language itself. There is also the standard library. Where possible, I employ the standard `string` type instead of using raw `char*` pointers, and I encourage you to do the same. `string` objects are no more difficult to manipulate than `char*`based strings, and they relieve you of most memory-management concerns. Furthermore, `string` objects are less susceptible to memory leaks if an exception is thrown (see Items 9 and 10). A well-implemented `string` type can hold its own in an efficiency contest with its `char*` equivalent, and it may even do better. (For insight into how this could be, see Item 29.) If you don't have access to an implementation of the standard `string` type, you almost certainly have access to *some* `string`-like class. Use it. Just about anything is preferable to raw `char*`s.

I use data structures from the standard library whenever I can. Such data structures are drawn from the Standard Template Library (the "STL" — see Item 35). The STL includes bitsets, vectors, lists, queues, stacks, maps, sets, and more, and you should prefer these standardized data structures to the ad hoc equivalents you might otherwise be tempted to write. Your compilers may not have the STL bundled in, but don't let that keep you from using it. Thanks to Silicon Graphics, you can download a free copy that works with many compilers from the SGI STL web site: `http://www.sgi.com/tech/stl/`.

If you currently use a library of algorithms and data structures and are happy with it, there's no need to switch to the STL just because it's "standard." However, if you have a choice between using an STL component or writing your own code from scratch, you should lean toward using the STL. Remember code reuse? STL (and the rest of the standard library) has lots of code that is very much worth reusing.

Conventions and Terminology

Any time I mention inheritance in this book, I mean public inheritance. If I don't mean public inheritance, I'll say so explicitly. When drawing inheritance hierarchies, I depict base-derived relationships by drawing arrows from derived classes to base classes. For example, here is a hierarchy from Item 31:



This notation is the reverse of the convention I employed in the first (but not the second) edition of *Effective C++*. I'm now convinced that most C++ practitioners draw inheritance arrows from derived to base classes, and I am happy to follow suit. Within such diagrams, abstract classes (e.g., `GameObject`) are shaded and concrete classes (e.g., `SpaceShip`) are unshaded.

Inheritance gives rise to pointers and references with two different types, a *static type* and a *dynamic type*. The static type of a pointer or reference is its *declared* type. The dynamic type is determined by the type of object it actually *refers* to. Here are some examples based on the classes above:

Click here to view code image

```
GameObject *pgo =                // static type of pgo is
  new SpaceShip;                 // GameObject*, dynamic
                                 // type is SpaceShip*

Asteroid *pa = new Asteroid;     // static type of pa is
                                 // Asteroid*. So is its
                                 // dynamic type

pgo = pa;                        // static type of pgo is
                                 // still (and always)
                                 // GameObject*. Its
                                 // dynamic type is now
                                 // Asteroid*
GameObject& rgo = *pa;           // static type of rgo is
                                 // GameObject, dynamic
                                 // type is Asteroid
```

These examples also demonstrate a naming convention I like. `pgo` is a pointer-to-`GameObject`; `pa` is a pointer-to-`Asteroid`; `rgo` is a reference-to-`GameObject`. I often concoct pointer and reference names in this fashion.

Two of my favorite parameter names are `lhs` and `rhs`, abbreviations for "left-hand side" and "right-hand side," respectively. To understand the rationale behind these names, consider a class for representing rational numbers:

```
class Rational { ... };
```

If I wanted a function to compare pairs of `Rational` objects, I'd declare it like this:

```
bool operator==(const Rational& lhs, const Rational& rhs);
```

That would let me write this kind of code:

```
Rational r1, r2;

...

if (r1 == r2) ...
```

Within the call to `operator==`, `r1` appears on the left-hand side of the "`==`" and is bound to `lhs`, while `r2` appears on the right-hand side of the "`==`" and is bound to `rhs`.

Other abbreviations I employ include *ctor* for "constructor," *dtor* for "destructor," and *RTTI* for C++'s support for runtime type identification (of which `dynamic_cast` is the most commonly used component).

When you allocate memory and fail to free it, you have a memory leak. Memory leaks arise in both C and C++, but in C++, memory leaks leak more than just memory. That's because C++ automatically calls constructors when objects are created, and constructors may themselves allocate resources. For example, consider this code:

Click here to view code image

```
class Widget { ... };           // some class – it doesn't
                                // matter what it is

Widget *pw = new Widget;        // dynamically allocate a
                                // Widget object

...                             // assume pw is never
                                // deleted
```

This code leaks memory, because the `Widget` pointed to by `pw` is never deleted. However, if the `Widget` constructor allocates additional resources that are to be released when the `Widget` is destroyed (such as file descriptors, semaphores, window handles, database locks, etc.), those resources are lost just as surely as the memory is. To emphasize that memory leaks in C++ often leak other resources, too, I usually speak of *resource leaks* in this book rather than memory leaks.

You won't see many inline functions in this book. That's not because I dislike inlining. Far from it, I believe that inline functions are an important feature of C++. However, the criteria for determining whether a function should be inlined can be complex, subtle, and platform-dependent. As a result, I avoid inlining unless there is a point about inlining I wish to make. When you see a non-inline function in *More Effective C++*, that doesn't mean I think it would be a bad idea to declare the function `inline`, it just means the decision to inline that function is independent of the material I'm examining at that point in the book.

A few C++ features have been *deprecated* by the standardization committee. Such features are slated for eventual removal from the language, because newer features have been added that do what the deprecated features do, but do it better. In this book, I identify deprecated constructs and explain what features replace them. You should try to avoid deprecated features where you can, but there's no reason to be overly concerned about their use. In the interest of preserving backward compatibility for their customers, compiler vendors are likely to support deprecated features for many years.

A *client* is somebody (a programmer) or something (a class or function, typically) that uses the code you write. For example, if you write a `Date` class (for representing birthdays, deadlines, when the Second Coming occurs, etc.), anybody using that class is your client. Furthermore, any sections of code that use the `Date` class are your clients as well. Clients are important. In fact, clients are the name of the game! If nobody uses the software you write, why write it? You will find I worry a lot about making things easier for clients, often at the expense of making things more difficult for you, because good software is "clientcentric" — it revolves around clients. If this strikes you as unreasonably philanthropic, view it instead through a lens of self-interest. Do you ever use the classes or functions you write? If so, you're your own client, so making things easier for clients in general also makes them easier for you.

When discussing class or function templates and the classes or functions generated from them, I reserve the right to be sloppy about the difference between the templates and their instantiations. For example, if `Array` is a class template taking a type parameter `T`, I may refer to a particular instantiation of the template as an `Array`, even though `Array<T>` is really the name of the class. Similarly, if `swap` is a function template taking a type parameter `T`, I may refer to an instantiation as `swap` instead of `swap<T>`. In cases where this kind of shorthand might be unclear, I include template parameters when referring to template instantiations.

Reporting Bugs, Making Suggestions, Getting Book Updates

I have tried to make this book as accurate, readable, and useful as possible, but I know there is room for improvement. If you find an error of any kind — technical, grammatical, typographical, *whatever* — please tell me about it. I will try to correct the mistake in future printings of the book, and if you are the first person to report it, I will gladly add your name to the book's acknowledgments. If you have other suggestions for improvement, I welcome those, too.

I continue to collect guidelines for effective programming in C++. If you have ideas for new guidelines, I'd be delighted if you'd share them with me. Send your guidelines, your comments, your criticisms, and your bug reports to:

Scott Meyers
c/o Editor-in-Chief, Corporate and Professional Publishing
Addison-Wesley Publishing Company
1 Jacob Way
Reading, MA 01867
U. S. A.

Alternatively, you may send electronic mail to `mec++@aristeia.com`.

I maintain a list of changes to this book since its first printing, including bug-fixes, clarifications, and technical updates. This list, along with other book-related information, is available from Addison-Wesley at World Wide Web URL `http://www.awl.com/cp/mec++.html`. It is also available via anonymous FTP from `ftp.awl.com` in the directory `cp/mec++`. If you would like a copy of the list of changes to this book, but you lack access to the Internet, please send a request to one of the addresses above, and I will see that the list is sent to you.

If you'd like to be notified when I make changes to this book, consider joining my mailing list. For details, consult `http://www.aristeia.com/MailingList/index.html`.

Enough preliminaries. On with the show!

Basics

Ah, the basics. Pointers, references, casts, arrays, constructors — you can't get much more basic than that. All but the simplest C++ programs use most of these features, and many programs use them all.

In spite of our familiarity with these parts of the language, sometimes they can still surprise us. This is especially true for programmers making the transition from C to C++, because the concepts behind references, dynamic casts, default constructors, and other non-C features are usually a little murky.

This chapter describes the differences between pointers and references and offers guidance on when to use each. It introduces the new C++ syntax for casts and explains why the new casts are superior to the C-style casts they replace. It examines the C notion of arrays and the C++ notion of polymorphism, and it describes why mixing the two is an idea whose time will never come. Finally, it considers the pros and cons of default constructors and suggests ways to work around language restrictions that encourage you to have one when none makes sense.

By heeding the advice in the items that follow, you'll make progress toward a worthy goal: producing software that expresses your design intentions clearly and correctly.

Item 1: Distinguish between pointers and references

Pointers and references *look* different enough (pointers use the "`*`" and "`->`" operators, references use "`.`"), but they seem to do similar things. Both pointers and references let you refer to other objects indirectly. How, then, do you decide when to use one and not the other?

First, recognize that there is no such thing as a null reference. A reference must *always* refer to some object. As a result, if you have a variable whose purpose is to refer to another object, but it is possible that there might not be an object to refer to, you should make the variable a pointer, because then you can set it to null. On the other hand, if the variable must *always* refer to an object, i.e., if your design does not allow for the possibility that the variable is null, you should probably make the variable a reference.

"But wait," you wonder, "what about underhandedness like this?"

Click here to view code image

```
char *pc = 0;                // set pointer to null

char& rc = *pc;              // make reference refer to
                             // dereferenced null pointer
```

Well, this is evil, pure and simple. The results are undefined (compilers can generate output to do anything they like), and people who write this kind of code should be shunned until they agree to cease and desist. If you have to worry about things like this in your software, you're probably best off avoiding references entirely. Either that or finding a better class of programmers to work with. We'll henceforth ignore the possibility that a reference can be "null."

Because a reference must refer to an object, C++ requires that references be initialized:

Click here to view code image

```
string& rs;                   // error! References must
                              // be initialized

string s("xyzzy");

string& rs = s;               // okay, rs refers to s
```

Pointers are subject to no such restriction:

Click here to view code image

```
string *ps;                   // uninitialized pointer:
                              // valid but risky
```

The fact that there is no such thing as a null reference implies that it can be more efficient to use references than to use pointers. That's because there's no need to test the validity of a reference before using it:

Click here to view code image

```
void printDouble(const double& rd)
{
   cout << rd;                 // no need to test rd; it
}                              // must refer to a double
```

Pointers, on the other hand, should generally be tested against null:

Click here to view code image

```
void printDouble(const double *pd)
{
  if (pd) {                    // check for null pointer
    cout << *pd;
  }
}
```

Another important difference between pointers and references is that pointers may be reassigned to refer to different objects. A reference, however, *always* refers to the object with which it is initialized:

Click here to view code image

```
string s1("Nancy");
string s2("Clancy");

string& rs = s1;                // rs refers to s1

string *ps = &s1;               // ps points to s1

rs = s2;                        // rs still refers to s1,
                                // but s1's value is now
                                // "Clancy"

ps = &s2;                       // ps now points to s2;
                                // s1 is unchanged
```

In general, you should use a pointer whenever you need to take into account the possibility that there's nothing to refer to (in which case you can set the pointer to null) or whenever you need to be able to refer to different things at different times (in which case you can change where the pointer points). You should use a reference whenever you know there will always be an object to refer to and you also know that once you're referring to that object, you'll never want to refer to anything else.

There is one other situation in which you should use a reference, and that's when you're implementing certain operators. The most common example is `operator[]`. This operator typically needs to return something that can be used as the target of an assignment:

Click here to view code image

```
vector<int> v(10);   // create an int vector of size 10;
                     // vector is a template in the
                     // standard C++ library (see Item 35)

v[5] = 10;           // the target of this assignment is
                     // the return value of operator[]
```

If `operator[]` returned a pointer, this last statement would have to be written this way:

```
*v[5] = 10;
```

But this makes it look like `v` is a vector of pointers, which it's not. For this reason, you'll almost always want `operator[]` to return a reference. (For an interesting exception to this rule, see Item 30.)

References, then, are the feature of choice when you *know* you have something to refer to and when you'll *never* want to refer to anything else. They're also appropriate when implementing operators whose syntactic requirements make the use of pointers undesirable. In all other cases, stick with pointers.

Item 2: Prefer C++-style casts

Consider the lowly cast. Nearly as much a programming pariah as the `goto`, it nonetheless endures, because when worse comes to worst and push comes to shove, casts can be necessary. Casts are especially necessary when worse comes to worst and push comes to shove.

Still, C-style casts are not all they might be. For one thing, they're rather crude beasts, letting you cast pretty much any type to pretty much any other type. It would be nice to be able to specify more precisely the purpose of each cast. There is a great difference, for example, between a cast that changes a pointer-to-`const`-object into a pointer-to-non-`const`-object (i.e., a cast that changes only the `const`ness of an object) and a cast that changes a pointer-to-base-class-object into a pointer-to-derived-class-object (i.e., a cast that completely changes an object's type). Traditional C-style casts make no such distinctions. (This is hardly a surprise. C-style casts were designed for

C, not C++.)

A second problem with casts is that they are hard to find. Syntactically, casts consist of little more than a pair of parentheses and an identifier, and parentheses and identifiers are used everywhere in C++. This makes it tough to answer even the most basic cast-related questions, questions like, "Are any casts used in this program?" That's because human readers are likely to overlook casts, and tools like `grep` cannot distinguish them from non-cast constructs that are syntactically similar.

C++ addresses the shortcomings of C-style casts by introducing four new cast operators, `static_cast`, `const_cast`, `dynamic_cast`, and `reinterpret_cast`. For most purposes, all you need to know about these operators is that what you are accustomed to writing like this,

```
(type) expression
```

you should now generally write like this:

```
static_cast<type>(expression)
```

For example, suppose you'd like to cast an `int` to a `double` to force an expression involving `int`s to yield a floating point value. Using C-style casts, you could do it like this:

Click here to view code image

```
int firstNumber, secondNumber;

...

double result = ((double)firstNumber)/secondNumber;
```

With the new casts, you'd write it this way:

```
double result = static_cast<double>
(firstNumber)/secondNumber;
```

Now *there's* a cast that's easy to see, both for humans and for programs.

`static_cast` has basically the same power and meaning as the general-purpose C-style cast. It also has the same kind of restrictions. For example, you can't cast a `struct` into an `int` or a `double` into a pointer using `static_cast` any more than you can with a C-style cast. Furthermore, `static_cast` can't remove `const`ness from an expression, because another new cast, `const_cast`, is designed specifically to do that.

The other new C++ casts are used for more restricted purposes. `const_cast` is used to cast away the `const`ness or `volatile`ness of an expression. By using a `const_cast`, you emphasize (to both humans and compilers) that the only thing you want to change through the cast is the `const`ness or `volatile`ness of something. This meaning is enforced by compilers. If you try to employ `const_cast` for anything other than modifying the `const`ness or `volatile`ness of an expression, your cast will be rejected. Here are some examples:

Click here to view code image

```
class Widget { ... };
class SpecialWidget: public Widget { ... };

void update(SpecialWidget *psw);

SpecialWidget sw;               // sw is a non-const
object,
const SpecialWidget& csw = sw;  // but csw is a reference
to
                                // it as a const object

update(&csw);         // error! can't pass a const
                      // SpecialWidget* to a function
                      // taking a SpecialWidget*

update(const_cast<SpecialWidget*>(&csw));
                      // fine, the constness of &csw is
                      // explicitly cast away (and
                      // csw - and sw - may now be
                      // changed inside update)

update((SpecialWidget*)&csw);
                      // same as above, but using a
                      // harder-to-recognize C-style cast

Widget *pw = new SpecialWidget;

update(pw);           // error! pw's type is Widget*, but
                      // update takes a SpecialWidget*

update(const_cast<SpecialWidget*>(pw));
                      // error! const_cast can be used only
                      // to affect constness or
```

```
volatileness,
                             // never to cast down the inheritance
                             // hierarchy
```

By far the most common use of `const_cast` is to cast away the `const`ness of an object.

The second specialized type of cast, `dynamic_cast`, is used to perform *safe casts* down or across an inheritance hierarchy. That is, you use `dynamic_cast` to cast pointers or references to base class objects into pointers or references to derived or sibling base class objects in such a way that you can determine whether the casts succeeded.[†] Failed casts are indicated by a null pointer (when casting pointers) or an exception (when casting references):

Click here to view code image

```
Widget *pw;

...

update(dynamic_cast<SpecialWidget*>(pw));
                            // fine, passes to update a pointer
                            // to the SpecialWidget pw points to
                            // if pw really points to one,
                            // otherwise passes the null pointer

void updateViaRef(SpecialWidget& rsw);

updateViaRef(dynamic_cast<SpecialWidget&>(*pw));
                            // fine, passes to updateViaRef the
                            // SpecialWidget pw points to if pw
                            // really points to one, otherwise
                            // throws an exception
```

`dynamic_cast`s are restricted to helping you navigate inheritance hierarchies. They cannot be applied to types lacking virtual functions (see also Item 24), nor can they cast away `const`ness:

Click here to view code image

```
int firstNumber, secondNumber;
...
double result = dynamic_cast<double>
(firstNumber)/secondNumber;
                              // error! int has no virtual
functions
```

```
const SpecialWidget sw;
...
update(dynamic_cast<SpecialWidget*>(&sw));
                     // error! dynamic_cast can't cast
                     // away constness
```

If you want to perform a cast on a type where inheritance is not involved, you probably want a `static_cast`. To cast `const`ness away, you always want a `const_cast`.

The last of the four new casting forms is `reinterpret_cast`. This operator is used to perform type conversions whose result is nearly always implementation-defined. As a result, `reinterpret_cast`s are rarely portable.

The most common use of `reinterpret_cast` is to cast between function pointer types. For example, suppose you have an array of pointers to functions of a particular type:

Click here to view code image

```
typedef void (*FuncPtr)();      // a FuncPtr is a pointer
                                // to a function taking no
                                // args and returning void

FuncPtr funcPtrArray[10];       // funcPtrArray is an array
                                // of 10 FuncPtrs
```

Let us suppose you wish (for some unfathomable reason) to place a pointer to the following function into `funcPtrArray`:

```
int doSomething();
```

You can't do what you want without a cast, because `doSomething` has the wrong type for `funcPtrArray`. The functions in `funcPtrArray` return `void`, but `doSomething` returns an `int`:

```
funcPtrArray[0] = &doSomething;     // error! type mismatch
```

A `reinterpret_cast` lets you force compilers to see things your way:

Click here to view code image

```
funcPtrArray[0] =                   // this compiles
  reinterpret_cast<FuncPtr>(&doSomething);
```

Casting function pointers is not portable (C++ offers no guarantee that all function pointers are represented the same way), and in some cases such casts yield incorrect results (see Item 31), so you should avoid casting function pointers unless your back's to the wall and a knife's at your throat. A sharp knife. A *very* sharp knife.

If your compilers lack support for the new casting forms, you can use traditional casts in place of `static_cast`, `const_cast`, and `reinterpret_cast`. Furthermore, you can use macros to approximate the new syntax:

Click here to view code image

```
#define static_cast(TYPE,EXPR)       ((TYPE)(EXPR))
#define const_cast(TYPE,EXPR)        ((TYPE)(EXPR))
#define reinterpret_cast(TYPE,EXPR)  ((TYPE)(EXPR))
```

You'd use the approximations like this:

Click here to view code image

```
double result = static_cast(double,
firstNumber)/secondNumber;

update(const_cast(SpecialWidget*, &sw));

funcPtrArray[0] = reinterpret_cast(FuncPtr, &doSomething);
```

These approximations won't be as safe as the real things, of course, but they will simplify the process of upgrading your code when your compilers support the new casts.

There is no easy way to emulate the behavior of a `dynamic_cast`, but many libraries provide functions to perform safe inheritance-based casts for you. If you lack such functions and you *must* perform this type of cast, you can fall back on C-style casts for those, too, but then you forego the ability to tell if the casts fail. Needless to say, you can define a macro to look like `dynamic_cast`, just as you can for the other casts:

```
#define dynamic_cast(TYPE,EXPR)     ((TYPE)(EXPR))
```

Remember that this approximation is not performing a true `dynamic_cast`; there is no way to tell if the cast fails.

I know, I know, the new casts are ugly and hard to type. If you find them too unpleasant to look at, take solace in the knowledge that C-style casts continue to be valid. However, what the new casts lack in beauty they make up for in precision of meaning and easy recognizability. Programs that use the new casts are easier to parse (both for humans and for tools), and they allow compilers to diagnose casting errors that would otherwise go undetected. These are powerful arguments for abandoning C-style casts, and there may also be a third: perhaps making casts ugly and hard to type is a *good* thing.

Item 3: Never treat arrays polymorphically

One of the most important features of inheritance is that you can manipulate derived class objects through pointers and references to base class objects. Such pointers and references are said to behave *polymorphically* — as if they had multiple types. C++ also allows you to manipulate *arrays* of derived class objects through base class pointers and references. This is no feature at all, because it almost never works the way you want it to.

For example, suppose you have a class `BST` (for binary search tree objects) and a second class, `BalancedBST`, that inherits from `BST`:

```
class BST { ... };

class BalancedBST: public BST { ... };
```

In a real program such classes would be templates, but that's unimportant here, and adding all the template syntax just makes things harder to read. For this discussion, we'll assume `BST` and `BalancedBST` objects contain only `int`s.

Consider a function to print out the contents of each `BST` in an array of `BST`s:

Click here to view code image

```
void printBSTArray(ostream& s,
                   const BST array[],
                   int numElements)
{
  for (int i = 0; i < numElements; ++i) {
    s << array[i];            // this assumes an
  }                           // operator<< is defined
}                             // for BST objects
```

This will work fine when you pass it an array of `BST` objects:

Click here to view code image

```
BST BSTArray[10];
```

```
...

printBSTArray(cout, BSTArray, 10);           // works fine
```

Consider, however, what happens when you pass `printBSTArray` an array of `BalancedBST` objects:

[Click here to view code image]

```
BalancedBST bBSTArray[10];

...

printBSTArray(cout, bBSTArray, 10);          // works fine?
```

Your compilers will accept this function call without complaint, but look again at the loop for which they must generate code:

[Click here to view code image]

```
for (int i = 0; i < numElements; ++i) {
  s << array[i];
}
```

Now, `array[i]` is really just shorthand for an expression involving pointer arithmetic: it stands for `*(array+i)`. We know that `array` is a pointer to the beginning of the array, but how far away from the memory location pointed to by `array` is the memory location pointed to by `array+i`? The distance between them is `i*sizeof(`*`an object in the array`*`)`, because there are `i` objects between `array[0]` and `array[i]`. In order for compilers to emit code that walks through the array correctly, they must be able to determine the size of the objects in the array. This is easy for them to do. The parameter `array` is declared to be of type array-of-`BST`, so each element of the array must be a `BST`, and the distance between `array` and `array+i` must be `i*sizeof(BST)`.

At least that's how your compilers look at it. But if you've passed an array of `BalancedBST` objects to `printBSTArray`, your compilers are probably wrong. In that case, they'd assume each object in the array is the size of a `BST`, but each object would actually be the size of a `BalancedBST`. Derived classes usually have more data members than their base classes, so derived class objects are usually larger than base class objects. We thus expect a `BalancedBST` object to be larger than a `BST` object. If it is, the pointer arithmetic generated for `printBSTArray` will be wrong for arrays of `BalancedBST` objects, and there's no telling what will happen when `printBSTArray` is invoked on a `BalancedBST` array. Whatever does happen, it's a good bet it won't be pleasant.

The problem pops up in a different guise if you try to delete an array of derived class objects through a base class pointer. Here's one way you might innocently attempt to do it:

Click here to view code image

```
// delete an array, but first log a message about its
// deletion
void deleteArray(ostream& logStream, BST array[])
{
  logStream << "Deleting array at address "
            << static_cast<void*>(array) << '\n';

  delete [] array;
}

BalancedBST *balTreeArray =          // create a BalancedBST
 new BalancedBST[50];                // array

...

deleteArray(cout, balTreeArray);    // log its deletion
```

You can't see it, but there's pointer arithmetic going on here, too. When an array is deleted, a destructor for each element of the array must be called (see Item 8). When compilers see the statement

```
delete [] array;
```

they must generate code that does something like this:

Click here to view code image

```
// destruct the objects in *array in the inverse order
// in which they were constructed
for (int i = the number of elements in the array - 1;
     i >= 0;
     --i)
  {
   array[i].BST::~BST();        // call array[i]'s
  }                             // destructor
```

Just as this kind of loop failed to work when you wrote it, it will fail to work when your compilers write it, too. The language specification says the result of deleting an array of derived class objects through a base class pointer is undefined, but we know what that really means: executing the code is almost certain to lead to grief. Polymorphism and pointer arithmetic simply don't mix. Array operations almost always involve pointer arithmetic, so arrays and polymorphism don't mix.

Note that you're unlikely to make the mistake of treating an array polymorphically if you avoid having a concrete class (like `BalancedBST`) inherit from another concrete class (such as `BST`). As Item 33 explains, designing your software so that concrete classes never inherit from one another has many benefits. I encourage you to turn to Item 33 and read all about them.

Item 4: Avoid gratuitous default constructors

A default constructor (i.e., a constructor that can be called with no arguments) is the C++ way of saying you can get something for nothing. Constructors initialize objects, so default constructors initialize objects without any information from the place where the object is being created. Sometimes this makes perfect sense. Objects that act like numbers, for example, may reasonably be initialized to zero or to undefined values. Objects that act like pointers (see Item 28) may reasonably be initialized to null or to undefined values. Data structures like linked lists, hash tables, maps, and the like may reasonably be initialized to empty containers.

Not all objects fall into this category. For many objects, there is no reasonable way to perform a complete initialization in the absence of outside information. For example, an object representing an entry in an address book makes no sense unless the name of the thing being entered is provided. In some companies, all equipment must be tagged with a corporate ID number, and creating an object to model a piece of equipment in such companies is nonsensical unless the appropriate ID number is provided.

In a perfect world, classes in which objects could reasonably be created from nothing would contain default constructors and classes in which information was required for object construction would not. Alas, ours is not the best of all possible worlds, so we must take additional concerns into account. In particular, if a class lacks a default constructor, there are restrictions on how you can use that class.

Consider a class for company equipment in which the corporate ID number of the equipment is a mandatory constructor argument:

Click here to view code image

```
class EquipmentPiece {
public:
  EquipmentPiece(int IDNumber);
  ...
};
```

Because `EquipmentPiece` lacks a default constructor, its use may be problematic in three contexts. The first is the creation of arrays. There is, in general, no way to specify constructor arguments for objects in arrays, so it is not usually possible to create arrays of `EquipmentPiece` objects:

Click here to view code image

```
EquipmentPiece bestPieces[10];      // error! No way to
call
                                    // EquipmentPiece ctors
EquipmentPiece *bestPieces =
  new EquipmentPiece[10];           // error! same problem
```

There are three ways to get around this restriction. A solution for non-heap arrays is to provide the necessary arguments at the point where the array is defined:

Click here to view code image

```
int ID1, ID2, ID3, ..., ID10;       // variables to hold
                                    // equipment ID numbers
...

EquipmentPiece bestPieces[] = {     // fine, ctor arguments
  EquipmentPiece(ID1),              // are provided
  EquipmentPiece(ID2),
  EquipmentPiece(ID3),
  ...,
  EquipmentPiece(ID10)
};
```

Unfortunately, there is no way to extend this strategy to heap arrays.

A more general approach is to use an array of *pointers* instead of an array of objects:

Click here to view code image

```
typedef EquipmentPiece* PEP;          // a PEP is a pointer
to
                                      // an EquipmentPiece

PEP bestPieces[10];                   // fine, no ctors
called

PEP *bestPieces = new PEP[10];        // also fine
```

Each pointer in the array can then be made to point to a different `EquipmentPiece` object:

Click here to view code image

```
for (int i = 0; i < 10; ++i)
  bestPieces[i] = new EquipmentPiece( ID Number );
```

There are two disadvantages to this approach. First, you have to remember to delete all the objects pointed to by the array. If you forget, you have a resource leak. Second, the total amount of memory you need increases, because you need the space for the pointers as well as the space for the `EquipmentPiece` objects.

You can avoid the space penalty if you allocate the raw memory for the array, then use "placement `new`" (see Item 8) to construct the `EquipmentPiece` objects in the memory:

Click here to view code image

```
// allocate enough raw memory for an array of 10
// EquipmentPiece objects; see Item 8 for details on
// the operator new[] function
void *rawMemory =
  operator new[](10*sizeof(EquipmentPiece));

// make bestPieces point to it so it can be treated as an
// EquipmentPiece array
EquipmentPiece *bestPieces =
  static_cast<EquipmentPiece*>(rawMemory);

// construct the EquipmentPiece objects in the memory
// using "placement new" (see Item 8)
for (int i = 0; i < 10; ++i)
  new (bestPieces+i) EquipmentPiece( ID Number );
```

Notice that you still have to provide a constructor argument for each `EquipmentPiece` object. This technique (as well as the array-of-pointers idea) allows you to create arrays of objects when a class lacks a default constructor; it doesn't show you how to bypass required constructor arguments. There is no way to do that. If there were, it would defeat the purpose of constructors, which is to *guarantee* that objects are initialized.

The downside to using placement `new`, aside from the fact that most programmers are unfamiliar with it (which will make maintenance more difficult), is that you must manually call destructors on the objects in the array when you want them to go out of existence, then you must manually deallocate the raw memory by calling `operator delete[]` (again, see Item 8):

Click here to view code image

```
// destruct the objects in bestPieces in the inverse
// order in which they were constructed
for (int i = 9; i >= 0; --i)
  bestPieces[i].~EquipmentPiece();

// deallocate the raw memory
operator delete[](rawMemory);
```

If you forget this requirement and use the normal array-deletion syntax, your program will behave unpredictably. That's because the result of deleting a pointer that didn't come from the `new` operator is undefined:

Click here to view code image

```
delete [] bestPieces;            // undefined! bestPieces
                                 // didn't come from the
new
                                 // operator
```

For more information on the `new` operator, placement `new` and how they interact with constructors and destructors, see Item 8.

The second problem with classes lacking default constructors is that they are ineligible for use with many template-based container classes. That's because it's a common requirement for such templates that the type used to instantiate the template provide a default constructor. This requirement almost always grows out of the fact that inside the template, an array of the template parameter type is being created. For example, a template for an `Array` class might look something like this:

Click here to view code image

```
template<class T>
class Array {
public:
  Array(int size);
  ...

private:
  T *data;
};

template<class T>
Array<T>::Array(int size)
{
  data = new T[size];            // calls T::T() for each
  ...                            // element of the array
}
```

In most cases, careful template design can eliminate the need for a default constructor. For example, the standard `vector` template (which generates classes that act like extensible arrays) has no requirement that its type parameter have a default constructor. Unfortunately, many templates are designed in a manner that is anything but careful. That being the case, classes without default constructors will be incompatible with many templates. As C++ programmers learn more about template design, this problem should recede in significance. How long it will take for that to happen, however, is anyone's guess.

The final consideration in the to-provide-a-default-constructor-or-not-to-provide-a-default-constructor dilemma has to do with virtual base classes. Virtual base classes lacking default constructors are a pain to work with. That's because the arguments for virtual base class constructors must be provided by the most derived class of the object being constructed. As a result, a virtual base class lacking a default constructor requires that *all* classes derived from that class — no matter how far removed — must know about, understand the meaning of, and provide for the virtual base class's constructors' arguments. Authors of derived classes neither expect nor appreciate this requirement.

Because of the restrictions imposed on classes lacking default constructors, some people believe *all* classes should have them, even if a default constructor doesn't have enough information to fully initialize objects of that class. For example, adherents to this philosophy might modify `EquipmentPiece` as follows:

Click here to view code image

```
class EquipmentPiece {
public:
  EquipmentPiece(int IDNumber = UNSPECIFIED);
  ...

private:
  static const int UNSPECIFIED;  // magic ID number value
                                 // meaning no ID was
};                               // specified
```

This allows `EquipmentPiece` objects to be created like this:

```
EquipmentPiece e;                // now okay
```

Such a transformation almost always complicates the other member functions of the class, because there is no longer any guarantee that the fields of an `EquipmentPiece` object have been meaningfully initialized. Assuming it makes no sense to have an `EquipmentPiece` without an ID field, most member functions must check to see if the ID is present. If it's not, they'll have to figure out how to stumble on anyway. Often it's not clear how to do that, and many implementations choose a solution that offers nothing but expediency: they throw an exception or they call a function that terminates the program. When that happens, it's difficult to argue that the overall quality of the software has been improved by including a default constructor in a class where none was warranted.

Inclusion of meaningless default constructors affects the efficiency of classes, too. If member functions have to test to see if fields have truly been initialized, clients of those functions have to pay for the time those tests take. Furthermore, they have to pay for the code that goes into those tests, because that makes executables and libraries bigger. They also have to pay for the code that handles the cases where the tests fail. All those costs are avoided if a class's constructors ensure that all fields of an object are correctly initialized. Often default constructors can't offer that kind of assurance, so it's best to avoid them in classes where they make no sense. That places some limits on how such classes can be used, yes, but it also guarantees that when you *do* use such classes, you can expect that the objects they generate are fully initialized and are efficiently implemented.

Operators

Overloadable operators — you gotta love 'em! They allow you to give your types the same syntax as C++'s built-in types, yet they let you put a measure of power into the functions *behind* the operators that's unheard of for the built-ins. Of course, the fact that you can make symbols like "`+`" and "`==`" do anything you want also means you can use overloaded operators to produce programs best described as impenetrable. Adept C++ programmers know how to harness the power of operator overloading without descending into the incomprehensible.

Regrettably, it is easy to make the descent. Single-argument constructors and implicit type conversion operators are particularly troublesome, because they can be invoked without there being any source code showing the calls. This can lead to program behavior that is difficult to understand. A different problem arises when you overload operators like `&&` and `||`, because the shift from built-in operator to user-defined function yields a subtle change in semantics that's easy to overlook. Finally, many operators are related to one another in standard ways, but the ability to overload operators makes it possible to violate the accepted relationships.

In the items that follow, I focus on explaining when and how overloaded operators are called, how they behave, how they should relate to one another, and how you can seize control of these aspects of overloaded operators. With the information in this chapter under your belt, you'll be overloading (or *not* overloading) operators like a pro.

Item 5: Be wary of user-defined conversion functions

C++ allows compilers to perform implicit conversions between types. In honor of its C heritage, for example, the language allows silent conversions from `char` to `int` and from `short` to `double`. This is why you can pass a `short` to a function that expects a `double` and still have the call succeed. The more frightening conversions in C — those that may lose information — are also present in C++, including conversion of `int` to `short` and `double` to (of all things) `char`.

You can't do anything about such conversions, because they're hardcoded into the language. When you add your own types, however, you have more control, because you can choose whether to provide the functions compilers are allowed to use for implicit type conversions.

Two kinds of functions allow compilers to perform such conversions: *single-argument constructors* and *implicit type conversion operators*. A single-argument constructor is a constructor that may be called with only one argument. Such a constructor may declare a single parameter or it may declare multiple parameters, with each parameter after the first having a default value. Here are two examples:

Click here to view code image

```
class Name {                        // for names of things
public:
  Name(const string& s);            // converts string to
                                    // Name
  ...

};

class Rational {                    // for rational numbers
public:
  Rational(int numerator = 0,       // converts int to
           int denominator = 1);    // Rational
  ...

};
```

An implicit type conversion operator is simply a member function with a strange-looking name: the word `operator` followed by a type specification. You aren't allowed to specify a type for the function's return value, because the type of the return value is basically just the name of the function. For example, to allow `Rational` objects to be implicitly converted to `double`s (which might be useful for mixed-mode arithmetic involving `Rational` objects), you might define class `Rational` like this:

Click here to view code image

```
class Rational {
public:
  ...
  operator double() const;      // converts Rational to
};                                     // double
```

This function would be automatically invoked in contexts like this:

Click here to view code image

```
Rational r(1, 2);                      // r has the value 1/2

double d = 0.5 * r;                    // converts r to a double,
                                       // then does multiplication
```

Perhaps all this is review. That's fine, because what I really want to explain is why you usually don't want to provide type conversion functions of *any* ilk.

The fundamental problem is that such functions often end up being called when you neither want nor expect them to be. The result can be incorrect and unintuitive program behavior that is maddeningly difficult to diagnose.

Let us deal first with implicit type conversion operators, as they are the easiest case to handle. Suppose you have a class for rational numbers similar to the one above, and you'd like to print `Rational` objects as if they were a built-in type. That is, you'd like to be able to do this:

```
Rational r(1, 2);

cout << r;                           // should print "1/2"
```

Further suppose you forgot to write an `operator<<` for `Rational` objects. You would probably expect that the attempt to print `r` would fail, because there is no appropriate `operator<<` to call. You would be mistaken. Your compilers, faced with a call to a function called `operator<<` that takes a `Rational`, would find that no such function existed, but they would then try to find an acceptable sequence of implicit type conversions they could apply to make the call succeed. The rules defining which sequences of conversions are acceptable are complicated, but in this case your compilers would discover they could make the call succeed by implicitly converting `r` to a `double` by calling `Rational::operator double`. The result of the code above would be to print `r` as a floating point number, not as a rational number. This is hardly a disaster, but it demonstrates the disadvantage of implicit type conversion operators: their presence can lead to the *wrong function* being called (i.e., one other than the one intended).

The solution is to replace the operators with equivalent functions that don't have the syntactically magic names. For example, to allow conversion of a

`Rational` object to a `double`, replace `operator double` with a function called something like `asDouble`:

Click here to view code image

```
class Rational {
public:
  ...
  double asDouble() const;          // converts Rational
};                                  // to double
```

Such a member function must be called explicitly:

Click here to view code image

```
Rational r(1, 2);

cout << r;                          // error! No operator<<
                                    // for Rationals

cout << r.asDouble();               // fine, prints r as a
                                    // double
```

In most cases, the inconvenience of having to call conversion functions explicitly is more than compensated for by the fact that unintended functions can no longer be silently invoked. In general, the more experience C++ programmers have, the more likely they are to eschew type conversion operators. The members of the committee working on the standard C++ library (see Item 35), for example, are among the most experienced in the business, and perhaps that's why the `string` type they added to the library contains no implicit conversion from a `string` object to a C-style `char*`. Instead, there's an explicit member function, `c_str`, that performs that conversion. Coincidence? I think not.

Implicit conversions via single-argument constructors are more difficult to eliminate. Furthermore, the problems these functions cause are in many cases worse than those arising from implicit type conversion operators.

As an example, consider a class template for array objects. These arrays allow clients to specify upper and lower index bounds:

Click here to view code image

```
template<class T>
class Array {
public:
    Array(int lowBound, int highBound);
    Array(int size);

    T& operator[](int index);

    ...

  };
```

The first constructor in the class allows clients to specify a range of array indices, for example, from 10 to 20. As a two-argument constructor, this function is ineligible for use as a type-conversion function. The second constructor, which allows clients to define `Array` objects by specifying only the number of elements in the array (in a manner similar to that used with built-in arrays), is different. It *can* be used as a type conversion function, and that can lead to endless anguish.

For example, consider a template specialization for comparing `Array<int>` objects and some code that uses such objects:

Click here to view code image

```
bool operator==(const Array<int>& lhs,
                const Array<int>& rhs);

Array<int> a(10);
Array<int> b(10);

...

for (int i = 0; i < 10; ++i)
  if (a == b[i]) {               // oops! "a" should be
"a[i]"
    do something for when
    a[i] and b[i] are equal;
  }
  else {
    do something for when they're not;
  }
```

We intended to compare each element of `a` to the corresponding element in `b`, but we accidentally omitted the subscripting syntax when we typed `a`. Certainly we expect this to elicit all manner of unpleasant commentary from our compilers, but they will complain not at all. That's because they see a call to `operator==` with arguments of type `Array<int>` (for `a`) and `int` (for `b[i]`), and though there is no `operator==` function taking those types, our compilers notice they can convert the `int` into an `Array<int>` object by calling the `Array<int>` constructor that takes a single `int` as an argument. This they proceed to do, thus generating code for a program we never meant to write, one that looks like this:

```
for (int i = 0; i < 10; ++i)
  if (a == static_cast< Array<int> >(b[i])) ...
```

Each iteration through the loop thus compares the contents of `a` with the contents of a temporary array of size `b[i]` (whose contents are presumably undefined). Not only is this unlikely to behave in a satisfactory manner, it is also tremendously inefficient, because each time through the loop we both create and destroy a temporary `Array<int>` object (see Item 19).

The drawbacks to implicit type conversion operators can be avoided by simply failing to declare the operators, but single-argument constructors cannot be so easily waved away. After all, you may really *want* to offer single-argument constructors to your clients. At the same time, you may wish to prevent compilers from calling such constructors indiscriminately. Fortunately, there is a way to have it all. In fact, there are two ways: the easy way and the way you'll have to use if your compilers don't yet support the easy way.

The easy way is to avail yourself of one of the newest C++ features, the `explicit` keyword. This feature was introduced specifically to address the problem of implicit type conversion, and its use is about as straightforward as can be. Constructors can be declared `explicit`, and if they are, compilers are prohibited from invoking them for purposes of implicit type conversion. Explicit conversions are still legal, however:

Click here to view code image

```
template<class T>
class Array {
public:
  ...
  explicit Array(int size);    // note use of "explicit"
  ...
};
```

```
Array<int> a(10);               // okay, explicit ctors can
                                // be used as usual for
                                // object construction

Array<int> b(10);               // also okay

if (a == b[i]) ...              // error! no way to
                                // implicitly convert
                                // int to Array<int>

if (a == Array<int>(b[i])) ... // okay, the conversion
                                // from int to Array<int> is
                                // explicit (but the logic
of
                                // the code is suspect)

if (a == static_cast< Array<int> >(b[i])) ...
                                // equally okay, equally
                                // suspect

if (a == (Array<int>)b[i]) ... // C-style casts are also
                                // okay, but the logic of
                                // the code is still suspect
```

In the example using `static_cast` (see Item 2), the space separating the two ">" characters is no accident. If the statement were written like this,

```
if (a == static_cast<Array<int>>(b[i])) ...
```

it would have a different meaning. That's because C++ compilers parse ">>" as a single token. Without a space between the ">" characters, the statement would generate a syntax error.

If your compilers don't yet support `explicit`, you'll have to fall back on home-grown methods for preventing the use of single-argument constructors as implicit type conversion functions. Such methods are obvious only *after* you've seen them.

I mentioned earlier that there are complicated rules governing which sequences of implicit type conversions are legitimate and which are not. One of those rules is that no sequence of conversions is allowed to contain more than one user-defined conversion (i.e., a call to a single-argument constructor or an implicit type conversion operator). By constructing your classes properly, you can take advantage of this rule so that the object constructions you want to allow are legal, but the implicit conversions you don't want to allow are illegal.

Consider the `Array` template again. You need a way to allow an integer specifying the size of the array to be used as a constructor argument, but you must at the same time prevent the implicit conversion of an integer into a temporary `Array` object. You accomplish this by first creating a new class, `ArraySize`. Objects of this type have only one purpose: they represent the size of an array that's about to be created. You then modify `Array`'s single-argument constructor to take an `ArraySize` object instead of an `int`. The code looks like this:

[Click here to view code image]

```
template<class T>
class Array {
public:

  class ArraySize {              // this class is new
  public:
    ArraySize(int numElements): theSize(numElements) {}
    int size() const { return theSize; }

  private:
    int theSize;
  };

  Array(int lowBound, int highBound);
  Array(ArraySize size);         // note new declaration

  ...

};
```

Here you've nested `ArraySize` inside `Array` to emphasize the fact that it's always used in conjunction with that class. You've also made `ArraySize` public in `Array` so that anybody can use it. Good.

Consider what happens when an `Array` object is defined via the class's single-argument constructor:

```
Array<int> a(10);
```

Your compilers are asked to call a constructor in the `Array<int>` class that takes an `int`, but there is no such constructor. Compilers realize they can convert the `int` argument into a temporary `ArraySize` object, and that `ArraySize` object is just what the `Array<int>` constructor needs, so compilers perform the conversion with their usual gusto. This allows the function call (and the attendant object construction) to succeed.

The fact that you can still construct `Array` objects with an `int` argument is reassuring, but it does you little good unless the type conversions you want to avoid are prevented. They are. Consider this code again:

Click here to view code image

```
bool operator==(const Array<int>& lhs,
                const Array<int>& rhs);

Array<int> a(10);
Array<int> b(10);

...

for (int i = 0; i < 10; ++i)
  if (a == b[i]) ...            // oops! "a" should be
"a[i]";
                                // this is now an error
```

Compilers need an object of type `Array<int>` on the right-hand side of the "`==`" in order to call `operator==` for `Array<int>` objects, but there is no single-argument constructor taking an `int` argument. Furthermore, compilers cannot consider converting the `int` into a temporary `ArraySize` object and then creating the necessary `Array<int>` object from this temporary, because that would call for two user-defined conversions, one from `int` to `ArraySize` and one from `ArraySize` to `Array<int>`. Such a conversion sequence is *verboten*, so compilers must issue an error for the code attempting to perform the comparison.

The use of the `ArraySize` class in this example might look like a special-purpose hack, but it's actually a specific instance of a more general technique. Classes like `ArraySize` are often called *proxy classes*, because each object of such a class stands for (is a proxy for) some other object. An `ArraySize` object is really just a stand-in for the integer used to specify the size of the `Array` being created. Proxy objects can give you control over aspects of your software's behavior — in this case implicit type conversions — that is otherwise beyond your grasp, so it's well worth your while to learn how to use them. How, you might wonder, can you acquire such learning? One way is to turn to Item 30; it's devoted to proxy classes.

Before you turn to proxy classes, however, reflect a bit on the lessons of this Item. Granting compilers license to perform implicit type conversions usually leads to more harm than good, so don't provide conversion functions unless you're *sure* you want them.

Item 6: Distinguish between prefix and postfix forms of increment and decrement operators

Long, long ago (the late '80s) in a language far, far away (C++ at that time), there was no way to distinguish between prefix and postfix invocations of the `++` and `--` operators. Programmers being programmers, they kvetched about this omission, and C++ was extended to allow overloading both forms of increment and decrement operators.

There was a syntactic problem, however, and that was that overloaded functions are differentiated on the basis of the parameter types they take, but neither prefix nor postfix increment or decrement takes an argument. To surmount this linguistic pothole, it was decreed that postfix forms take an `int` argument, and compilers silently pass 0 as that `int` when those functions are called:

Click here to view code image

```
class UPInt {                      // "unlimited precision
int"
public:
  UPInt& operator++();             // prefix ++
  const UPInt operator++(int);     // postfix ++

  UPInt& operator--();             // prefix --
  const UPInt operator--(int);     // postfix --

  UPInt& operator+=(int);          // a += operator for UPInts
                                   // and ints
  ...

};

UPInt i;
```

```
++i;                              // calls i.operator++();
i++;                              // calls i.operator++(0);

--i;                              // calls i.operator--();
i--;                              // calls i.operator--(0);
```

This convention is a little on the odd side, but you'll get used to it. More important to get used to, however, is this: the prefix and postfix forms of these operators return *different types*. In particular, prefix forms return a reference, postfix forms return a `const` object. We'll focus here on the prefix and postfix ++ operators, but the story for the – operators is analogous.

From your days as a C programmer, you may recall that the prefix form of the increment operator is sometimes called "increment and fetch," while the postfix form is often known as "fetch and increment." These two phrases are important to remember, because they all but act as formal specifications for how prefix and postfix increment should be implemented:

Click here to view code image

```
// prefix form: increment and fetch
UPInt& UPInt::operator++()
{
  *this += 1;                       // increment
  return *this;                     // fetch
}

// postfix form: fetch and increment
const UPInt UPInt::operator++(int)
{
  const UPInt oldValue = *this;     // fetch
  ++(*this);                        // increment

  return oldValue;                  // return what was
}                                   // fetched
```

Note how the postfix operator makes no use of its parameter. This is typical. The only purpose of the parameter is to distinguish prefix from postfix function invocation. Many compilers issue warnings if you fail to use named parameters in the body of the function to which they apply, and this can be annoying. To avoid such warnings, a common strategy is to omit names for parameters you don't plan to use; that's what's been done above.

It's clear why postfix increment must return an object (it's returning an old value), but why a `const` object? Imagine that it did not. Then the following would be legal:

Click here to view code image

```
UPInt i;

i++++;                              // apply postfix increment
                                    // twice
```

This is the same as

```
i.operator++(0).operator++(0);
```

and it should be clear that the second invocation of `operator++` is being applied to the object returned from the first invocation.

There are two reasons to abhor this. First, it's inconsistent with the behavior of the built-in types. A good rule to follow when designing classes is *when in doubt, do as the* `int`*s do*, and the `int`s most certainly do not allow double application of postfix increment:

```
int i;

i++++;                              // error!
```

The second reason is that double application of postfix increment almost never does what clients expect it to. As noted above, the second application of `operator++` in a double increment changes the value of the object returned from the first invocation, *not* the value of the original object. Hence, if

```
i++++;
```

were legal, `i` would be incremented only once. This is counterintuitive and confusing (for both `int`s and `UPInt`s), so it's best prohibited.

C++ prohibits it for `int`s, but you must prohibit it yourself for classes you write. The easiest way to do this is to make the return type of postfix increment a `const` object. Then when compilers see

```
i++++;              // same as i.operator++(0).operator++(0);
```

they recognize that the `const` object returned from the first call to `operator++` is being used to call `operator++` again. `operator++`, however, is a non-`const` member function, so `const` objects — such as those returned from postfix `operator++` — can't call it.[‡] If you've ever wondered if it makes sense to have functions return `const` objects, now you know: sometimes it does, and postfix increment and decrement are examples.

If you're the kind who worries about efficiency, you probably broke into a sweat when you first saw the postfix increment function. That function has to create a temporary object for its return value (see Item 19), and the implementation above also creates an explicit temporary object (`oldValue`) that has to be constructed and destructed. The prefix increment function has no such temporaries. This leads to the possibly startling conclusion that, for efficiency reasons alone, clients of `UPInt` should prefer prefix increment to postfix increment unless they really need the behavior of postfix increment. Let us be explicit about this. When dealing with user-defined types, prefix increment should be used whenever possible, because it's inherently more efficient.

Let us make one more observation about the prefix and postfix increment operators. Except for their return values, they do the same thing: they increment a value. That is, they're *supposed* to do the same thing. How can you be sure the behavior of postfix increment is consistent with that of prefix increment? What guarantee do you have that their implementations won't diverge over time, possibly as a result of different programmers maintaining and enhancing them? Unless you've followed the design principle embodied by the code above, you have no such guarantee. That principle is that postfix increment and decrement should be implemented *in terms of* their prefix counterparts. You then need only maintain the prefix versions, because the postfix versions will automatically behave in a consistent fashion.

As you can see, mastering prefix and postfix increment and decrement is easy. Once you know their proper return types and that the postfix operators should be implemented in terms of the prefix operators, there's very little more to learn.

Item 7: Never overload `&&`, `||`, or `,`

Like C, C++ employs short-circuit evaluation of boolean expressions. This means that once the truth or falsehood of an expression has been determined, evaluation of the expression ceases, even if some parts of the expression haven't yet been examined. For example, in this case,

```
char *p;

...

if ((p != 0) && (strlen(p) > 10)) ...
```

there is no need to worry about invoking `strlen` on `p` if it's a null pointer, because if the test of `p` against 0 fails, `strlen` will never be called. Similarly, given

Click here to view code image

```
int rangeCheck(int index)
{
  if ((index < lowerBound) || (index > upperBound)) ...

  ...

}
```

`index` will never be compared to `upperBound` if it's less than `lowerBound`.

This is the behavior that has been drummed into C and C++ programmers since time immemorial, so this is what they expect. Furthermore, they write programs whose correct behavior *depends* on short-circuit evaluation. In the first code fragment above, for example, it is important that `strlen` not be invoked if `p` is a null pointer, because the standard for C++ states (as does the standard for C) that the result of invoking `strlen` on a null pointer is undefined.

C++ allows you to customize the behavior of the `&&` and `||` operators for user-defined types. You do it by overloading the functions `operator&&` and `operator||`, and you can do this at the global scope or on a per-class basis. If you decide to take advantage of this opportunity, however, you must be aware that you are changing the rules of the game quite radically, because you are replacing short-circuit semantics with *function call* semantics. That is, if you overload `operator&&`, what looks to you like this,

```
if (expression1 && expression2) ...
```

looks to compilers like one of these:

Click here to view code image

```
if (expression1.operator&&(expression2)) ...
                               // when operator&& is a
                               // member function

if (operator&&(expression1, expression2)) ...
                               // when operator&& is a
                               // global function
```

This may not seem like that big a deal, but function call semantics differ from short-circuit semantics in two crucial ways. First, when a function call is made, *all* parameters must be evaluated, so when calling the functions `operator&&` and `operator||`, *both* parameters are evaluated. There is, in other words, no short circuit. Second, the language specification leaves undefined the order of evaluation of parameters to a function call, so there is no way of knowing whether `expression1` or `expression2` will be evaluated first. This stands in stark contrast to short-circuit evaluation, which *always* evaluates its arguments in left-to-right order.

As a result, if you overload `&&` or `||`, there is no way to offer programmers the behavior they both expect and have come to depend on. So don't overload `&&` or `||`.

The situation with the comma operator is similar, but before we delve into that, I'll pause and let you catch the breath you lost when you gasped, "The comma operator? There's a *comma* operator?" There is indeed.

The comma operator is used to form *expressions*, and you're most likely to run across it in the update part of a `for` loop. The following function, for example, is based on one in the second edition of Kernighan's and Ritchie's classic *The C Programming Language* (Prentice-Hall, 1988):

Click here to view code image

```
// reverse string s in place
void reverse(char s[])
{

  for (int i = 0, j = strlen(s)-1;
       i < j;
       ++i, --j)               // aha! the comma operator!
  {
    int c = s[i];
    s[i] = s[j];
    s[j] = c;
    }
}
```

Here, `i` is incremented and `j` is decremented in the final part of the `for` loop. It is convenient to use the comma operator here, because only an expression is valid in the final part of a `for` loop; separate statements to change the values of `i` and `j` would be illegal.

Just as there are rules in C++ defining how `&&` and `||` behave for built-in types, there are rules defining how the comma operator behaves for such types. An expression containing a comma is evaluated by first evaluating the part of the expression to the left of the comma, then evaluating the expression to the right of the comma; the result of the overall comma expression is the value of the expression on the right. So in the final part of the loop above, compilers first evaluate `++i`, then `--j`, and the result of the comma expression is the value returned from `--j`.

Perhaps you're wondering why you need to know this. You need to know because you need to mimic this behavior if you're going to take it upon yourself to write your own comma operator. Unfortunately, you can't perform the requisite mimicry.

If you write `operator,` as a non-member function, you'll never be able to guarantee that the left-hand expression is evaluated before the right-hand expression, because both expressions will be passed as arguments in a function call (to `operator,`). But you have no control over the order in which a function's arguments are evaluated. So the non-member approach is definitely out.

That leaves only the possibility of writing `operator,` as a member function. Even here you can't rely on the left-hand operand to the comma operator being evaluated first, because compilers are not constrained to do things that way. Hence, you can't overload the comma operator and also guarantee it will behave the way it's supposed to. It therefore seems imprudent to overload it at all.

You may be wondering if there's an end to this overloading madness. After all, if you can overload the comma operator, what *can't* you overload? As it turns out, there are limits. You can't overload the following operators:

Click here to view code image

```
.            .*           ::           ?:

new          delete       sizeof       typeid

static_cast  dynamic_cast const_cast   reinterpret_cast
```

You can overload these:

```
operator new        operator delete
operator new[]      operator delete[]
+     -      *      /     %      ^      &      |      ~
!     =      <      >     +=     -=     *=     /=     %=
^=    &=     |=     <<    >>     >>=    <<=    ==     !=
<=    >=     &&     ||    ++     --     ,      ->*    ->
()    []
```

(For information on the `new` and `delete` operators, as well as `operator new`, `operator delete`, `operator new[]`, and `operator delete[]`, see Item 8.)

Of course, just because you can overload these operators is no reason to run off and do it. The purpose of operator overloading is to make programs easier to read, write, and understand, not to dazzle others with your knowledge that comma is an operator. If you don't have a good reason for overloading an operator, don't overload it. In the case of `&&`, `||`, and `,`, it's difficult to have a good reason, because no matter how hard you try, you can't make them behave the way they're supposed to.

Item 8: Understand the different meanings of `new` and `delete`

It occasionally seems as if people went out of their way to make C++ terminology difficult to understand. Case in point: the difference between the *`new` operator* and *`operator new`*.

When you write code like this,

```
string *ps = new string("Memory Management");
```

the `new` you are using is the `new` operator. This operator is built into the language and, like `sizeof`, you can't change its meaning: it always does the same thing. What it does is twofold. First, it allocates enough memory to hold an object of the type requested. In the example above, it allocates enough memory to hold a `string` object. Second, it calls a constructor to initialize an object in the memory that was allocated. The `new` operator always does those two things; you can't change its behavior in any way.

What you can change is *how* the memory for an object is allocated. The `new` operator calls a function to perform the requisite memory allocation, and you can rewrite or overload that function to change its behavior. The name of the function the `new` operator calls to allocate memory is `operator new`. Honest.

The `operator new` function is usually declared like this:

```
void * operator new(size_t size);
```

The return type is `void*`, because this function returns a pointer to raw, uninitialized memory. (If you like, you can write a version of `operator new` that initializes the memory to some value before returning a pointer to it, but this is not commonly done.) The `size_t` parameter specifies how much memory to allocate. You can overload `operator new` by adding additional parameters, but the first parameter must always be of type `size_t`.

You'll probably never want to call `operator new` directly, but on the off chance you do, you'll call it just like any other function:

```
void *rawMemory = operator new(sizeof(string));
```

Here `operator new` will return a pointer to a chunk of memory large enough to hold a `string` object.

Like `malloc`, `operator new`'s only responsibility is to allocate memory. It knows nothing about constructors. All `operator new` understands is memory allocation. It is the job of the `new` operator to take the raw memory that `operator new` returns and transform it into an object. When your compilers see a statement like

```
string *ps = new string("Memory Management");
```

they must generate code that more or less corresponds to this:

Click here to view code image

```
void *memory =                              // get raw memory
  operator new(sizeof(string));             // for a string
                                            // object

call string::string("Memory Management")    // initialize the
on *memory;                                 // object in the
                                            // memory
```

```
string *ps =                              // make ps point
to
  static_cast<string*>(memory);           // the new object
```

Notice that the second step above involves calling a constructor, something you, a mere programmer, are prohibited from doing. Your compilers are unconstrained by mortal limits, however, and they can do whatever they like. That's why you must use the `new` operator if you want to conjure up a heap-based object: you can't directly call the constructor necessary to initialize the object (including such crucial components as its vtbl — see Item 24).

Placement `new`

There are times when you really *want* to call a constructor directly. Invoking a constructor on an existing object makes no sense, because constructors initialize objects, and an object can only be initialized — given its first value — once. But occasionally you have some raw memory that's already been allocated, and you need to construct an object in the memory you have. A special version of `operator new` called *placement `new`* allows you to do it.

As an example of how placement `new` might be used, consider this:

Click here to view code image

```
class Widget {
public:
  Widget(int widgetSize);
  ...
};

Widget * constructWidgetInBuffer(void *buffer,
                                 int widgetSize)
{
  return new (buffer) Widget(widgetSize);
}
```

This function returns a pointer to a `Widget` object that's constructed *within* the buffer passed to the function. Such a function might be useful for applications using shared memory or memory-mapped I/O, because objects in such applications must be placed at specific addresses or in memory allocated by special routines. (For a different example of how placement `new` can be used, see Item 4.)

Inside `constructWidgetInBuffer`, the expression being returned is

```
new (buffer) Widget(widgetSize)
```

This looks a little strange at first, but it's just a use of the `new` operator in which an additional argument (`buffer`) is being specified for the implicit call that the `new` operator makes to `operator new`. The `operator new` thus called must, in addition to the mandatory `size_t` argument, accept a `void*` parameter that points to the memory the object being constructed is to occupy. That `operator new` *is* placement `new`, and it looks like this:

Click here to view code image

```
void * operator new(size_t, void *location)
{
  return location;
}
```

This is probably simpler than you expected, but this is all placement `new` needs to do. After all, the purpose of `operator new` is to find memory for an object and return a pointer to that memory. In the case of placement `new`, the caller already knows what the pointer to the memory should be, because the caller knows where the object is supposed to be placed. All placement `new` has to do, then, is return the pointer that's passed into it. (The unused (but mandatory) `size_t` parameter has no name to keep compilers from complaining about its not being used; see Item 6.) Placement `new` is part of the standard C++ library. To use placement `new`, all you have to do is `#include <new>` (or, if your compilers don't yet support the new-style header names, `<new.h>`).

If we step back from placement `new` for a moment, we'll see that the relationship between the `new` operator and `operator new`, though perhaps terminologically confusing, is conceptually straightforward. If you want to create an object on the heap, use the `new` operator. It both allocates memory and calls a constructor for the object. If you only want to allocate memory, call `operator new`; no constructor will be called. If you want to customize the memory allocation that takes place when heap objects are created, write your own version of `operator new` and use the `new` operator; it will automatically invoke your custom version of `operator new`. If you want to construct an object in memory you've already got a pointer to, use placement `new`.

Deletion and Memory Deallocation

To avoid resource leaks, every dynamic allocation must be matched by an equal and opposite deallocation. The function `operator delete` is to the built-in `delete` operator as `operator new` is to the `new` operator. When you say something like this,

Click here to view code image

```
string *ps;
...
delete ps;                          // use the delete
operator
```

your compilers must generate code both to destruct the object `ps` points to and to deallocate the memory occupied by that object.

The memory deallocation is performed by the `operator delete` function, which is usually declared like this:

```
void operator delete(void *memoryToBeDeallocated);
```

Hence,

```
delete ps;
```

causes compilers to generate code that approximately corresponds to this:

Click here to view code image

```
ps->~string();                    // call the object's dtor

operator delete(ps);              // deallocate the memory
                                  // the object occupied
```

One implication of this is that if you want to deal only with raw, uninitialized memory, you should bypass the `new` and `delete` operators entirely. Instead, you should call `operator new` to get the memory and `operator delete` to return it to the system:

Click here to view code image

```
void *buffer =                         // allocate enough
  operator new(50*sizeof(char));       // memory to hold 50
                                       // chars; call no ctors

...
```

```
operator delete(buffer);           // deallocate the
memory;
                                   // call no dtors
```

This is the C++ equivalent of calling `malloc` and `free`.

If you use placement `new` to create an object in some memory, you should avoid using the `delete` operator on that memory. That's because the `delete` operator calls `operator delete` to deallocate the memory, but the memory containing the object wasn't allocated by `operator new` in the first place; placement `new` just returned the pointer that was passed to it. Who knows where that pointer came from? Instead, you should undo the effect of the constructor by explicitly calling the object's destructor:

Click here to view code image

```
// functions for allocating and deallocating memory in
// shared memory
void * mallocShared(size_t size);
void freeShared(void *memory);

void *sharedMemory = mallocShared(sizeof(Widget));

Widget *pw =                                      // as above,
  constructWidgetInBuffer(sharedMemory, 10);  // placement
                                              // new is
used

...

delete pw;           // undefined! sharedMemory came from
                     // mallocShared, not operator new

pw->~Widget();       // fine, destructs the Widget pointed
to
                     // by pw, but doesn't deallocate the
                     // memory containing the Widget

freeShared(pw);      // fine, deallocates the memory pointed
                     // to by pw, but calls no destructor
```

As this example demonstrates, if the raw memory passed to placement `new` was itself dynamically allocated (through some unconventional means), you must still deallocate that memory if you wish to avoid a memory leak.

Arrays

So far so good, but there's farther to go. Everything we've examined so far concerns itself with only one object at a time. What about array allocation? What happens here?

Click here to view code image

```
string *ps = new string[10];     // allocate an array of
                                 // objects
```

The `new` being used is still the `new` operator, but because an array is being created, the `new` operator behaves slightly differently from the case of single-object creation. For one thing, memory is no longer allocated by `operator new`. Instead, it's allocated by the array-allocation equivalent, a function called `operator new[]` (often referred to as "array `new`.") Like `operator new`, `operator new[]` can be overloaded. This allows you to seize control of memory allocation for arrays in the same way you can control memory allocation for single objects.

(`operator new[]` is a relatively recent addition to C++, so your compilers may not support it yet. If they don't, the global version of `operator new` will be used to allocate memory for every array, regardless of the type of objects in the array. Customizing array-memory allocation under such compilers is daunting, because it requires that you rewrite the global `operator new`. This is not a task to be undertaken lightly. By default, the global `operator new` handles *all* dynamic memory allocation in a program, so any change in its behavior has a dramatic and pervasive effect. Furthermore, there is only one global `operator new` with the "normal" signature (i.e., taking the single `size_t` parameter), so if you decide to claim it as your own, you instantly render your software incompatible with any library that makes the same decision. (See also Item 27.) As a result of these considerations, custom memory management for arrays is not usually a reasonable design decision for compilers lacking support for `operator new[]`.)

The second way in which the `new` operator behaves differently for arrays than for objects is in the number of constructor calls it makes. For arrays, a constructor must be called for *each object* in the array:

Click here to view code image

```
string *ps =                // call operator new[] to allocate
```

```
   new string[10];         // memory for 10 string objects,
                           // then call the default string
                           // ctor for each array element
```

Similarly, when the `delete` operator is used on an array, it calls a destructor for each array element and then calls `operator delete[]` to deallocate the memory:

Click here to view code image

```
 delete [] ps;             // call the string dtor for each
                           // array element, then call
                           // operator delete[] to
                           // deallocate the array's memory
```

Just as you can replace or overload `operator delete`, you can replace or overload `operator delete[]`. There are some restrictions on how they can be overloaded, however; consult a good C++ text for details. (For ideas on good C++ texts, see the recommendations beginning on page 285B.)

So there you have it. The `new` and `delete` operators are built-in and beyond your control, but the memory allocation and deallocation functions they call are not. When you think about customizing the behavior of the `new` and `delete` operators, remember that you can't really do it. You can modify *how* they do what they do, but *what* they do is fixed by the language.

Exceptions

The addition of exceptions to C++ changes things. Profoundly. Radically. Possibly uncomfortably. The use of raw, unadorned pointers, for example, becomes risky. Opportunities for resource leaks increase in number. It becomes more difficult to write constructors and destructors that behave the way we want them to. Special care must be taken to prevent program execution from abruptly halting. Executables and libraries typically increase in size and decrease in speed.

And these are just the things we know. There is much the C++ community does not know about writing programs using exceptions, including, for the most part, how to do it correctly. There is as yet no agreement on a body of techniques that, when applied routinely, leads to software that behaves predictably and reliably when exceptions are thrown. (For insight into some of the issues involved, see the article by Tom Cargill I refer to on page 287B.)

We do know this much: programs that behave well in the presence of exceptions do so because they were *designed* to, not because they happen to. Exception-safe programs are not created by accident. The chances of a program behaving well in the presence of exceptions when it was not designed for exceptions are about the same as the chances of a program behaving well in the presence of multiple threads of control when it was not designed for multi-threaded execution: about zero.

That being the case, why use exceptions? Error codes have sufficed for C programmers ever since C was invented, so why mess with exceptions, especially if they're as problematic as I say? The answer is simple: exceptions cannot be ignored. If a function signals an exceptional condition by setting a status variable or returning an error code, there is no way to guarantee the function's caller will check the variable or examine the code. As a result, execution may continue long past the point where the condition was encountered. If the function signals the condition by throwing an exception, however, and that exception is not caught, program execution immediately ceases.

This is behavior that C programmers can approach only by using `setjmp` and `longjmp`. But `longjmp` exhibits a serious deficiency when used with C++: it fails to call destructors for local objects when it adjusts the stack. Most C++ programs depend on such destructors being called, so `setjmp` and `longjmp` make a poor substitute for true exceptions. If you need a way of signaling exceptional conditions that cannot be ignored, and if you must ensure that local destructors are called when searching the stack for code that can handle exceptional conditions, you need C++ exceptions. It's as simple as that.

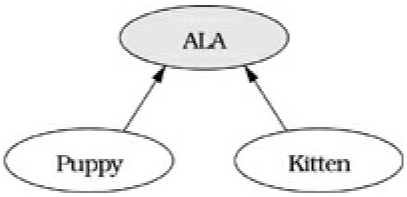
Because we have much to learn about programming with exceptions, the Items that follow comprise an incomplete guide to writing exception-safe software. Nevertheless, they introduce important considerations for anyone using exceptions in C++. By heeding the guidance in the material below, you'll improve the correctness, robustness, and efficiency of the software you write, and you'll sidestep many problems that commonly arise when working with exceptions.

Item 9: Use destructors to prevent resource leaks

Say good-bye to pointers. Admit it: you never really liked them that much anyway.

Okay, you don't have to say good-bye to *all* pointers, but you do need to say *sayonara* to pointers that are used to manipulate local resources. Suppose, for example, you're writing software at the Shelter for Adorable Little Animals, an organization that finds homes for puppies and kittens. Each day the shelter creates a file containing information on the adoptions it arranged that day, and your job is to write a program to read these files and do the appropriate processing for each adoption.

A reasonable approach to this task is to define an abstract base class, `ALA` ("Adorable Little Animal"), plus concrete derived classes for puppies and kittens. A virtual function, `processAdoption`, handles the necessary species-specific processing:



[Click here to view code image](#)

```
class ALA {
public:
  virtual void processAdoption() = 0;
  ...
};

class Puppy: public ALA {
public:
  virtual void processAdoption();
  ...
};

class Kitten: public ALA {
public:
  virtual void processAdoption();
  ...
};
```

You'll need a function that can read information from a file and produce either a `Puppy` object or a `Kitten` object, depending on the information in the file. This is a perfect job for a *virtual constructor*, a kind of function described in [Item 25](#). For our purposes here, the function's declaration is all we need:

[Click here to view code image](#)

```
// read animal information from s, then return a pointer
// to a newly allocated object of the appropriate type
ALA * readALA(istream& s);
```

The heart of your program is likely to be a function that looks something like this:

[Click here to view code image](#)

```
void processAdoptions(istream& dataSource)
{
  while (dataSource) {                  // while there's data
    ALA *pa = readALA(dataSource);      // get next animal
    pa->processAdoption();              // process adoption
    delete pa;                          // delete object that
  }                                     // readALA returned
}
```

This function loops through the information in `dataSource`, processing each entry as it goes. The only mildly tricky thing is the need to remember to delete `pa` at the end of each iteration. This is necessary because `readALA` creates a new heap object each time it's called. Without the call to `delete`, the loop would contain a resource leak.

Now consider what would happen if `pa->processAdoption` threw an exception. `processAdoptions` fails to catch exceptions, so the exception would propagate to `processAdoptions`'s caller. In doing so, all statements in `processAdoptions` after the call to `pa->processAdoption` would be skipped, and that means `pa` would never be deleted. As a result, anytime `pa->processAdoption` throws an exception, `processAdoptions` contains a resource leak.

Plugging the leak is easy enough,

Click here to view code image

```
void processAdoptions(istream& dataSource)
{
  while (dataSource) {
    ALA *pa = readALA(dataSource);

    try {
      pa->processAdoption();
    }
    catch (...) {           // catch all exceptions

      delete pa;            // avoid resource leak when an
                            // exception is thrown

      throw;                // propagate exception to caller
    }

    delete pa;              // avoid resource leak when no
  }                         // exception is thrown
}
```

but then you have to litter your code with `try` and `catch` blocks. More importantly, you are forced to duplicate cleanup code that is common to both normal and exceptional paths of control. In this case, the call to `delete` must be duplicated. Like all replicated code, this is annoying to write and difficult to maintain, but it also *feels wrong*. Regardless of whether we leave `processAdoptions` by a normal return or by throwing an exception, we need to delete `pa`, so why should we have to say that in more than one place?

We don't have to if we can somehow move the cleanup code that must always be executed into the destructor for an object local to `processAdoptions`. That's because local objects are always destroyed when leaving a function, regardless of how that function is exited. (The only exception to this rule is when you call `longjmp`, and this shortcoming of `longjmp` is the primary reason why C++ has support for exceptions in the first place.) Our real concern, then, is moving the `delete` from `processAdoptions` into a destructor for an object local to `processAdoptions`.

The solution is to replace the pointer `pa` with an object that *acts like* a pointer. That way, when the pointer-like object is (automatically) destroyed, we can have its destructor call `delete`. Objects that act like pointers, but do more, are called *smart pointers*, and, as Item 28 explains, you can make pointer-like objects very smart indeed. In this case, we don't need a particularly brainy pointer, we just need a pointer-like object that knows enough to delete what it points to when the pointer-like object goes out of scope.

It's not difficult to write a class for such objects, but we don't need to. The standard C++ library contains a class template called `auto_ptr` that does just what we want. Each `auto_ptr` class takes a pointer to a heap object in its constructor and deletes that object in its destructor. Boiled down to these essential functions, `auto_ptr` looks like this:

Click here to view code image

```
template<class T>
class auto_ptr {
public:
  auto_ptr(T *p = 0): ptr(p) {}     // save ptr to object
  ~auto_ptr() { delete ptr; }       // delete ptr to object

private:
  T *ptr;                           // raw ptr to object
};
```

The standard version of `auto_ptr` is much fancier, and this stripped-down implementation isn't suitable for real use[†] (we must add at least the copy constructor, assignment operator, and pointer-emulating functions discussed in Item 28), but the concept behind it should be clear: use `auto_ptr` objects instead of raw pointers, and you won't have to worry about heap objects not being deleted, not even when exceptions are thrown. (Because the `auto_ptr` destructor uses the single-object form of `delete`, `auto_ptr` is not suitable for use with pointers to *arrays* of objects. If you'd like an `auto_ptr`-like template for arrays, you'll have to write your own. In such cases, however, it's often a better design decision to use a `vector` instead of an array, anyway.)

Using an `auto_ptr` object instead of a raw pointer, `processAdoptions` looks like this:

Click here to view code image

```
void processAdoptions(istream& dataSource)
{
  while (dataSource) {
    auto_ptr<ALA> pa(readALA(dataSource));
    pa->processAdoption();
  }
}
```

This version of `processAdoptions` differs from the original in only two ways. First, `pa` is declared to be an `auto_ptr<ALA>` object, not a raw `ALA*` pointer. Second, there is no `delete` statement at the end of the loop. That's it. Everything else is identical, because, except for destruction, `auto_ptr` objects act just like normal pointers. Easy, huh?

The idea behind `auto_ptr` — using an object to store a resource that needs to be automatically released and relying on that object's destructor to release it — applies to more than just pointer-based resources. Consider a function in a GUI application that needs to create a window to display some information:

Click here to view code image

```
// this function may leak resources if an exception
// is thrown
void displayInfo(const Information& info)
{
  WINDOW_HANDLE w(createWindow());

  display info in window corresponding to w;

  destroyWindow(w);
}
```

Many window systems have C-like interfaces that use functions like `createWindow` and `destroyWindow` to acquire and release window resources. If an exception is thrown during the process of displaying `info` in `w`, the window for which `w` is a handle will be lost just as surely as any other dynamically allocated resource.

The solution is the same as it was before. Create a class whose constructor and destructor acquire and release the resource:

Click here to view code image

```
// class for acquiring and releasing a window handle
class WindowHandle {
public:
  WindowHandle(WINDOW_HANDLE handle): w(handle) {}
  ~WindowHandle() { destroyWindow(w); }

  operator WINDOW_HANDLE() { return w; }     // see below

private:
  WINDOW_HANDLE w;

  // The following functions are declared private to
prevent
  // multiple copies of a WINDOW_HANDLE from being created.
  // See Item 28 for a discussion of a more flexible
approach.
  WindowHandle(const WindowHandle&);
  WindowHandle& operator=(const WindowHandle&);
};
```

This looks just like the `auto_ptr` template, except that assignment and copying are explicitly prohibited, and there is an implicit conversion operator that can be used to turn a `WindowHandle` into a `WINDOW_HANDLE`. This capability is essential to the practical application of a `WindowHandle` object, because it means you can use a `WindowHandle` just about anywhere you would normally use a raw `WINDOW_HANDLE`. (See Item 5, however, for why you should generally be leery of implicit type conversion operators.)

Given the `WindowHandle` class, we can rewrite `displayInfo` as follows:

Click here to view code image

```
// this function avoids leaking resources if an
// exception is thrown
void displayInfo(const Information& info)
{
  WindowHandle w(createWindow());
```

```
    display info in window corresponding to w;

  }
```

Even if an exception is thrown within `displayInfo`, the window created by `createWindow` will always[†] be destroyed.

By adhering to the rule that resources should be encapsulated inside objects, you can usually avoid resource leaks in the presence of exceptions. But what happens if an exception is thrown while you're in the process of acquiring a resource, e.g., while you're in the constructor of a resource-acquiring class? What happens if an exception is thrown during the automatic destruction of such resources? Don't constructors and destructors call for special techniques? They do, and you can read about them in Items 10 and 11.

Item 10: Prevent resource leaks in constructors

Imagine you're developing software for a multimedia address book. Such an address book might hold, in addition to the usual textual information of a person's name, address, and phone numbers, a picture of the person and the sound of their voice (possibly giving the proper pronunciation of their name).

To implement the book, you might come up with a design like this:

Click here to view code image

```
class Image {                        // for image data
public:
  Image(const string& imageDataFileName);
  ...
};

class AudioClip {                    // for audio data
public:
  AudioClip(const string& audioDataFileName);
  ...
};

class PhoneNumber { ... };           // for holding phone
numbers

class BookEntry {                   // for each entry in the
public:                             // address book
```

```
  BookEntry(const string& name,
            const string& address = "",
            const string& imageFileName = "",
            const string& audioClipFileName = "");
  ~BookEntry();

  // phone numbers are added via this function
  void addPhoneNumber(const PhoneNumber& number);
  ...

private:
  string theName;                // person's name
  string theAddress;             // their address
  list<PhoneNumber> thePhones;  // their phone numbers
  Image *theImage;               // their image
  AudioClip *theAudioClip;       // an audio clip from them
};
```

Each `BookEntry` must have name data, so you require that as a constructor argument (see Item 4), but the other fields — the person's address and the names of files containing image and audio data — are optional. Note the use of the `list` class to hold the person's phone numbers. This is one of several container classes that are part of the standard C++ library (see Item 35).

A straightforward way to write the `BookEntry` constructor and destructor is as follows:

Click here to view code image

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(0), theAudioClip(0)
{
  if (imageFileName != "") {
    theImage = new Image(imageFileName);
  }

  if (audioClipFileName != "") {
    theAudioClip = new AudioClip(audioClipFileName);
  }
}
```

```
BookEntry::~BookEntry()
{
  delete theImage;
  delete theAudioClip;
}
```

The constructor initializes the pointers `theImage` and `theAudioClip` to null, then makes them point to real objects if the corresponding arguments are non-empty strings. The destructor deletes both pointers, thus ensuring that a `BookEntry` object doesn't give rise to a resource leak. Because C++ guarantees it's safe to delete null pointers, `BookEntry`'s destructor need not check to see if the pointers actually point to something before deleting them.

Everything looks fine here, and under normal conditions everything is fine, but under abnormal conditions — under *exceptional* conditions — things are not fine at all.

Consider what will happen if an exception is thrown during execution of this part of the `BookEntry` constructor:

Click here to view code image

```
if (audioClipFileName != "") {
  theAudioClip = new AudioClip(audioClipFileName);
}
```

An exception might arise because `operator new` (see Item 8) is unable to allocate enough memory for an `AudioClip` object. One might also arise because the `AudioClip` constructor itself throws an exception. Regardless of the cause of the exception, if one is thrown within the `BookEntry` constructor, it will be propagated to the site where the `BookEntry` object is being created.

Now, if an exception is thrown during creation of the object `theAudioClip` is supposed to point to (thus transferring control out of the `BookEntry` constructor), who deletes the object that `theImage` already points to? The obvious answer is that `BookEntry`'s destructor does, but the obvious answer is wrong. `BookEntry`'s destructor will never be called. Never.

C++ destroys only *fully constructed* objects, and an object isn't fully constructed until its constructor has run to completion. So if a `BookEntry` object `b` is created as a local object,

Click here to view code image

```
void testBookEntryClass()
{
  BookEntry b("Addison-Wesley Publishing Company",
              "One Jacob Way, Reading, MA 01867");
  ...
}
```

and an exception is thrown during construction of `b`, `b`'s destructor will not be called. Furthermore, if you try to take matters into your own hands by allocating `b` on the heap and then calling `delete` if an exception is thrown,

[Click here to view code image]

```
void testBookEntryClass()
{
  BookEntry *pb = 0;

  try {
    pb = new BookEntry("Addison-Wesley Publishing Company",
                       "One Jacob Way, Reading, MA 01867");
    ...
  }
  catch (...) {                   // catch all exceptions

    delete pb;                    // delete pb when an
                                  // exception is thrown

    throw;                        // propagate exception to
  }                               // caller

  delete pb;                      // delete pb normally
}
```

you'll find that the `Image` object allocated inside `BookEntry`'s constructor is still lost, because no assignment is made to `pb` unless the `new` operation succeeds. If `BookEntry`'s constructor throws an exception, `pb` will be the null pointer, so deleting it in the `catch` block does nothing except make you feel better about yourself. Using the smart pointer class `auto_ptr<BookEntry>` (see Item 9) instead of a raw `BookEntry*` won't do you any good either, because the assignment to `pb` still won't be made unless the `new` operation succeeds.

There is a reason why C++ refuses to call destructors for objects that haven't been fully constructed, and it's not simply to make your life more difficult. It's because it would, in many cases, be a nonsensical thing — possibly a harmful thing — to do. If a destructor were invoked on an object that wasn't fully constructed, how would the destructor know what to do? The only way it could know would be if bits had been added to each object indicating how much of the constructor had been executed. Then the destructor could check the bits and (maybe) figure out what actions to take. Such bookkeeping would slow down constructors, and it would make each object larger, too. C++ avoids this overhead, but the price you pay is that partially constructed objects aren't automatically destroyed.

Because C++ won't clean up after objects that throw exceptions during construction, you must design your constructors so that they clean up after themselves. Often, this involves simply catching all possible exceptions, executing some cleanup code, then rethrowing the exception so it continues to propagate. This strategy can be incorporated into the `BookEntry` constructor like this:

Click here to view code image

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(0), theAudioClip(0)
{
  try {                          // this try block is new
    if (imageFileName != "") {
      theImage = new Image(imageFileName);
    }

    if (audioClipFileName != "") {
      theAudioClip = new AudioClip(audioClipFileName);
    }
  }
  catch (...) {                  // catch any exception
    delete theImage;             // perform necessary
    delete theAudioClip;         // cleanup actions

    throw;                       // propagate the exception
  }
}
```

There is no need to worry about `BookEntry`'s non-pointer data members. Data members are automatically initialized before a class's constructor is called, so if a `BookEntry` constructor body begins executing, the object's `theName`, `theAddress`, and `thePhones` data members have already been fully constructed. As fully constructed objects, these data members will be automatically destroyed even if an exception arises in the `BookEntry` constructor[†]. Of course, if these objects' constructors call functions that might throw exceptions, *those* constructors have to worry about catching the exceptions and performing any necessary cleanup before allowing them to propagate.

You may have noticed that the statements in `BookEntry`'s `catch` block are almost the same as those in `BookEntry`'s destructor. Code duplication here is no more tolerable than it is anywhere else, so the best way to structure things is to move the common code into a private helper function and have both the constructor and the destructor call it:

Click here to view code image

```
class BookEntry {
public:
  ...                           // as before
private:
  ...
  void cleanup();               // common cleanup statements
};

void BookEntry::cleanup()
{
  delete theImage;
  delete theAudioClip;
}

BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(0), theAudioClip(0)
{
  try {
    ...                          // as before
  }
  catch (...) {
    cleanup();                   // release resources
    throw;                         // propagate exception
  }
}

BookEntry::~BookEntry()
{
  cleanup();
}
```

This is nice, but it doesn't put the topic to rest. Let us suppose we design our `BookEntry` class slightly differently so that `theImage` and `theAudioClip` are *constant* pointers:

Click here to view code image

```
class BookEntry {
public:
  ...                                   // as above

private:
  ...
  Image * const theImage;               // pointers are now
  AudioClip * const theAudioClip;       // const
};
```

Such pointers must be initialized via the member initialization lists of `BookEntry`'s constructors, because there is no other way to give `const` pointers a value. A common temptation is to initialize `theImage` and `theAudioClip` like this,

Click here to view code image

```
// an implementation that may leak resources if an
// exception is thrown
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
:theName(name), theAddress(address),
 theImage(imageFileName != ""
          ? new Image(imageFileName)
          : 0),
 theAudioClip(audioClipFileName != ""
              ? new AudioClip(audioClipFileName)
              : 0)
{}
```

but this leads to the problem we originally wanted to eliminate: if an exception is thrown during initialization of `theAudioClip`, the object pointed to by `theImage` is never destroyed. Furthermore, we can't solve the problem by adding `try` and `catch` blocks to the constructor, because `try` and `catch` are statements, and member initialization lists allow only expressions. (That's why we had to use the `?:` syntax instead of the `if-then-else` syntax in the initialization of `theImage` and `theAudioClip`.)

Nevertheless, the only way to perform cleanup chores before exceptions propagate out of a constructor is to catch those exceptions, so if we can't put `try` and `catch` in a member initialization list, we'll have to put them somewhere else. One possibility is inside private member functions that return pointers with which `theImage` and `theAudioClip` should be initialized:

[Click here to view code image]

```
class BookEntry {
public:
  ...                                // as above

private:
  ...                                // data members as above

  Image * initImage(const string& imageFileName);
  AudioClip * initAudioClip(const string&
                            audioClipFileName);
};

BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
:theName(name), theAddress(address),
 theImage(initImage(imageFileName)),
 theAudioClip(initAudioClip(audioClipFileName))
{}
// theImage is initialized first, so there is no need to
// worry about a resource leak if this initialization
// fails. This function therefore handles no exceptions
Image * BookEntry::initImage(const string& imageFileName)
{
  if (imageFileName != "") return new Image(imageFileName);
  else return 0;
}
```

```
// theAudioClip is initialized second, so it must make
// sure theImage's resources are released if an exception
// is thrown during initialization of theAudioClip. That's
// why this function uses try...catch.
AudioClip * BookEntry::initAudioClip(const string&
                                     audioClipFileName)
{
  try {
    if (audioClipFileName != "") {
      return new AudioClip(audioClipFileName);
    }
    else return 0;
  }
  catch (...) {
    delete theImage;
    throw;
  }
}
```

This is perfectly kosher, and it even solves the problem we've been laboring to overcome. The drawback is that code that conceptually belongs in a constructor is now dispersed across several functions, and that's a maintenance headache.

A better solution is to adopt the advice of Item 9 and treat the objects pointed to by `theImage` and `theAudioClip` as resources to be managed by local objects. This solution takes advantage of the facts that both `theImage` and `theAudioClip` are pointers to dynamically allocated objects and that those objects should be deleted when the pointers themselves go away. This is precisely the set of conditions for which the `auto_ptr` classes (see Item 9) were designed. We can therefore change the raw pointer types of `theImage` and `theAudioClip` to their `auto_ptr` equivalents:

Click here to view code image

```
class BookEntry {
public:
  ...                                         // as above

private:
  ...
  const auto_ptr<Image> theImage;             // these are now
  const auto_ptr<AudioClip> theAudioClip;     // auto_ptr
objects
};
```

Doing this makes `BookEntry`'s constructor leak-safe in the presence of exceptions, and it lets us initialize `theImage` and `theAudioClip` using the member initialization list:

Click here to view code image

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(imageFileName != ""
           ? new Image(imageFileName)
           : 0),
  theAudioClip(audioClipFileName != ""
               ? new AudioClip(audioClipFileName)
               : 0)
{}
```

In this design, if an exception is thrown during initialization of `theAudioClip`, `theImage` is already a fully constructed object, so it will automatically be destroyed, just like `theName`, `theAddress`, and `thePhones`. Furthermore, because `theImage` and `theAudioClip` are now objects, they'll be destroyed automatically when the `BookEntry` object containing them is. Hence there's no need to manually delete what they point to. That simplifies `BookEntry`'s destructor considerably:

Click here to view code image

```
BookEntry::~BookEntry()
{}                              // nothing to do!
```

This means you could eliminate `BookEntry`'s destructor entirely.

It all adds up to this: if you replace pointer class members with their corresponding `auto_ptr` objects, you fortify your constructors against resource leaks in the presence of exceptions, you eliminate the need to manually deallocate resources in destructors, and you allow `const` member pointers to be handled in the same graceful fashion as non-`const` pointers.

Dealing with the possibility of exceptions during construction can be tricky, but `auto_ptr` (and `auto_ptr`-like classes) can eliminate most of the drudgery. Their use leaves behind code that's not only easy to understand, it's robust in the face of exceptions, too.

Item 11: Prevent exceptions from leaving destructors

There are two situations in which a destructor is called. The first is when an object is destroyed under "normal" conditions, e.g., when it goes out of scope or is explicitly `delete`d. The second is when an object is destroyed by the exception-handling mechanism during the stack-unwinding part of exception propagation.

That being the case, an exception may or may not be active when a destructor is invoked. Regrettably, there is no way to distinguish between these conditions from inside a destructor.[†] As a result, you must write your destructors under the conservative assumption that an exception *is* active, because if control leaves a destructor due to an exception while another exception is active, C++ calls the `terminate` function. That function does just what its name suggests: it terminates execution of your program. Furthermore, it terminates it *immediately*; not even local objects are destroyed.

As an example, consider a `Session` class for monitoring on-line computer sessions, i.e., things that happen from the time you log in through the time you log out. Each `Session` object notes the date and time of its creation and destruction:

Click here to view code image

```
class Session {
public:
  Session();
  ~Session();
  ...

private:
  static void logCreation(Session *objAddr);
  static void logDestruction(Session *objAddr);
};
```

The functions `logCreation` and `logDestruction` are used to record object creations and destructions, respectively. We might therefore expect that we could code `Session`'s destructor like this:

```
Session::~Session()
{
  logDestruction(this);
}
```

This looks fine, but consider what would happen if `logDestruction` throws an exception. The exception would not be caught in `Session`'s destructor, so it would be propagated to the caller of that destructor. But if the destructor was itself being called because some other exception had been thrown, the `terminate` function would automatically be invoked, and that would stop your program dead in its tracks.

In many cases, this is not what you'll want to have happen. It may be unfortunate that the `Session` object's destruction can't be logged, it might even be a major inconvenience, but is it really so horrific a prospect that the program can't continue running? If not, you'll have to prevent the exception thrown by `logDestruction` from propagating out of `Session`'s destructor. The only way to do that is by using `try` and `catch` blocks. A naive attempt might look like this,

Click here to view code image

```
Session::~Session()
{
  try {
    logDestruction(this);
  }
  catch (...) {
    cerr << "Unable to log destruction of Session object "
         << "at address "
         << this
         << ".\n";
  }
}
```

but this is probably no safer than our original code. If one of the calls to `operator<<` in the `catch` block results in an exception being thrown, we're back where we started, with an exception leaving the `Session` destructor.

We could always put a `try` block inside the `catch` block, but that seems a bit extreme. Instead, we'll just forget about logging `Session` destructions if `logDestruction` throws an exception:

```
Session::~Session()
```

```
{
  try {
    logDestruction(this);
  }
  catch (...) {}
}
```

The `catch` block appears to do nothing, but appearances can be deceiving. That block prevents exceptions thrown from `logDestruction` from propagating beyond `Session`'s destructor. That's all it needs to do. We can now rest easy knowing that if a `Session` object is destroyed as part of stack unwinding, `terminate` will not be called.

There is a second reason why it's bad practice to allow exceptions to propagate out of destructors. If an exception is thrown from a destructor and is not caught there, that destructor won't run to completion. (It will stop at the point where the exception is thrown.) If the destructor doesn't run to completion, it won't do everything it's supposed to do. For example, consider a modified version of the `Session` class where the creation of a session starts a database transaction and the termination of a session ends that transaction:

Click here to view code image

```
Session::Session()              // to keep things simple,
{                               // this ctor handles no
                                // exceptions
  logCreation(this);
  startTransaction();           // start DB transaction
}

Session::~Session()
{
  logDestruction(this);
  endTransaction();             // end DB transaction
}
```

Here, if `logDestruction` throws an exception, the transaction started in the `Session` constructor will never be ended. In this case, we might be able to reorder the function calls in `Session`'s destructor to eliminate the problem, but if `endTransaction` might throw an exception, we've no choice but to revert to `try` and `catch` blocks.

We thus find ourselves with two good reasons for keeping exceptions from propagating out of destructors. First, it prevents `terminate` from being called during the stack-unwinding part of exception propagation. Second, it helps ensure that destructors always accomplish everything they are supposed to accomplish. Each argument is convincing in its own right, but together, the case is ironclad.

Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function

The syntax for declaring function parameters is almost the same as that for `catch` clauses:

Click here to view code image

```
class Widget { ... };            // some class; it makes no
                                 // difference what it is

void f1(Widget w);               // all these functions
void f2(Widget& w);              // take parameters of
void f3(const Widget& w);        // type Widget, Widget&, or
void f4(Widget *pw);             // Widget*
void f5(const Widget *pw);

catch (Widget w) ...             // all these catch clauses
catch (Widget& w) ...            // catch exceptions of
catch (const Widget& w) ...      // type Widget, Widget&, or
catch (Widget *pw) ...           // Widget*
catch (const Widget *pw) ...
```

You might therefore assume that passing an exception from a `throw` site to a `catch` clause is basically the same as passing an argument from a function call site to the function's parameter. There are some similarities, to be sure, but there are significant differences, too.

Let us begin with a similarity. You can pass both function parameters and exceptions by value, by reference, or by pointer. What *happens* when you pass parameters and exceptions, however, is quite different. This difference grows out of the fact that when you call a function, control eventually returns to the call site (unless the function fails to return), but when you throw an exception, control does *not* return to the `throw` site.

Consider a function that both passes a `Widget` as a parameter and throws a `Widget` as an exception:

[Click here to view code image]()

```
// function to read the value of a Widget from a stream
istream operator>>(istream& s, Widget& w);

void passAndThrowWidget()
{
  Widget localWidget;

  cin >> localWidget;     // pass localWidget to operator>>

  throw localWidget;      // throw localWidget as an
exception
}
```

When `localWidget` is passed to `operator>>`, no copying is performed. Instead, the reference `w` inside `operator>>` is bound to `localWidget`, and anything done to `w` is really done to `localWidget`. It's a different story when `localWidget` is thrown as an exception. Regardless of whether the exception is caught by value or by reference (it can't be caught by pointer — that would be a type mismatch), a copy of `localWidget` will be made, and it is the *copy* that is passed to the `catch` clause. This must be the case, because `localWidget` will go out of scope once control leaves `passAndThrowWidget`, and when `localWidget` goes out of scope, its destructor will be called. If `localWidget` itself were passed to a `catch` clause, the clause would receive a destructed `Widget`, an ex-`Widget`, a former `Widget`, the carcass of what once was but is no longer a `Widget`. That would not be useful, and that's why C++ specifies that an object thrown as an exception is copied.

This copying occurs even if the object being thrown is not in danger of being destroyed. For example, if `passAndThrowWidget` declares `localWidget` to be static,

[Click here to view code image]()

```
void passAndThrowWidget()
{
  static Widget localWidget;    // this is now static; it
                                // will exist until the
                                // end of the program
  cin >> localWidget;           // this works as before

  throw localWidget;            // a copy of localWidget is
}                               // still made and thrown
```

a copy of `localWidget` would still be made when the exception was thrown.

This means that even if the exception is caught by reference, it is not possible for the `catch` block to modify `localWidget`; it can only modify a *copy* of `localWidget`. This mandatory copying of exception objects[†] helps explain another difference between parameter passing and throwing an exception: the latter is typically much slower than the former (see Item 15).

When an object is copied for use as an exception, the copying is performed by the object's copy constructor. This copy constructor is the one in the class corresponding to the object's *static* type, not its dynamic type. For example, consider this slightly modified version of `passAndThrowWidget`:

Click here to view code image

```
class Widget { ... };
class SpecialWidget: public Widget { ... };

void passAndThrowWidget()
{
  SpecialWidget localSpecialWidget;

  ...

  Widget& rw = localSpecialWidget;  // rw refers to a
                                    // SpecialWidget

  throw rw;                         // this throws an
                                    // exception of type
}                                   // Widget!
```

Here a `Widget` exception is thrown, even though `rw` refers to a `SpecialWidget`. That's because `rw`'s static type is `Widget`, not `SpecialWidget`. That `rw` actually refers to a `SpecialWidget` is of no concern to your compilers; all they care about is `rw`'s static type. This behavior may not be what you want, but it's consistent with all other cases in which C++ copies objects. Copying is always based on an object's static type (but see Item 25 for a technique that lets you make copies on the basis of an object's dynamic type).

The fact that exceptions are copies of other objects has an impact on how you propagate exceptions from a `catch` block. Consider these two `catch` blocks, which at first glance appear to do the same thing:

Click here to view code image

```
catch (Widget& w)        // catch Widget exceptions
```

```
{
  ...                        // handle the exception

  throw;                     // rethrow the exception so it
}                            // continues to propagate

catch (Widget& w)            // catch Widget exceptions
{
  ...                        // handle the exception

  throw w;                   // propagate a copy of the
}                            // caught exception
```

The only difference between these blocks is that the first one rethrows the current exception, while the second one throws a new copy of the current exception. Setting aside the performance cost of the additional copy operation, is there a difference between these approaches?

There is. The first block rethrows the *current* exception, regardless of its type. In particular, if the exception originally thrown was of type `SpecialWidget`, the first block would propagate a `SpecialWidget` exception, even though `w`'s static type is `Widget`. This is because no copy is made when the exception is rethrown. The second `catch` block throws a *new* exception, which will always be of type `Widget`, because that's `w`'s static type. In general, you'll want to use the

```
throw;
```

syntax to rethrow the current exception, because there's no chance that that will change the type of the exception being propagated. Furthermore, it's more efficient, because there's no need to generate a new exception object.

(Incidentally, the copy made for an exception is a *temporary* object. As Item 19 explains, this gives compilers the right to optimize it out of existence. I wouldn't expect your compilers to work that hard, however. Exceptions are supposed to be rare, so it makes little sense for compiler vendors to pour a lot of energy into their optimization.)

Let us examine the three kinds of `catch` clauses that could catch the `Widget` exception thrown by `passAndThrowWidget`. They are:

Click here to view code image

```
catch (Widget w) ...            // catch exception by value
catch (Widget& w) ...          // catch exception by
                               // reference

catch (const Widget& w) ...    // catch exception by
                               // reference-to-const
```

Right away we notice another difference between parameter passing and exception propagation. A thrown object (which, as explained above, is always a temporary) may be caught by simple reference; it need not be caught by reference-to-`const`. Passing a temporary object to a non-`const` reference parameter is not allowed for function calls (see Item 19), but it is for exceptions.

Let us overlook this difference, however, and return to our examination of copying exception objects. We know that when we pass a function argument by value, we make a copy of the passed object, and we store that copy in a function parameter. The same thing happens when we pass an exception by value. Thus, when we declare a `catch` clause like this,

```
catch (Widget w) ...           // catch by value
```

we expect to pay for the creation of *two* copies of the thrown object, one to create the temporary that all exceptions generate, the second to copy that temporary into `w`. Similarly, when we catch an exception by reference,

Click here to view code image

```
catch (Widget& w) ...          // catch by reference

catch (const Widget& w) ...    // also catch by reference
```

we still expect to pay for the creation of a copy of the exception: the copy that is the temporary. In contrast, when we pass function parameters by reference, no copying takes place. When throwing an exception, then, we expect to construct (and later destruct) one more copy of the thrown object than if we passed the same object to a function.

We have not yet discussed throwing exceptions by pointer, but throw by pointer is equivalent to pass by pointer. Either way, a copy of the *pointer* is passed. About all you need to remember is not to throw a pointer to a local object, because that local object will be destroyed when the exception leaves the local object's scope. The `catch` clause would then be initialized with a pointer to an object that had already been destroyed. This is the behavior the mandatory copying rule is designed to avoid.

The way in which objects are moved from call or `throw` sites to parameters or `catch` clauses is one way in which argument passing differs from exception propagation. A second difference lies in what constitutes a type match between caller or thrower and callee or catcher. Consider the `sqrt` function from the standard math library:

```
double sqrt(double);            // from <cmath> or <math.h>
```

We can determine the square root of an integer like this:

```
int i;
double sqrtOfi = sqrt(i);
```

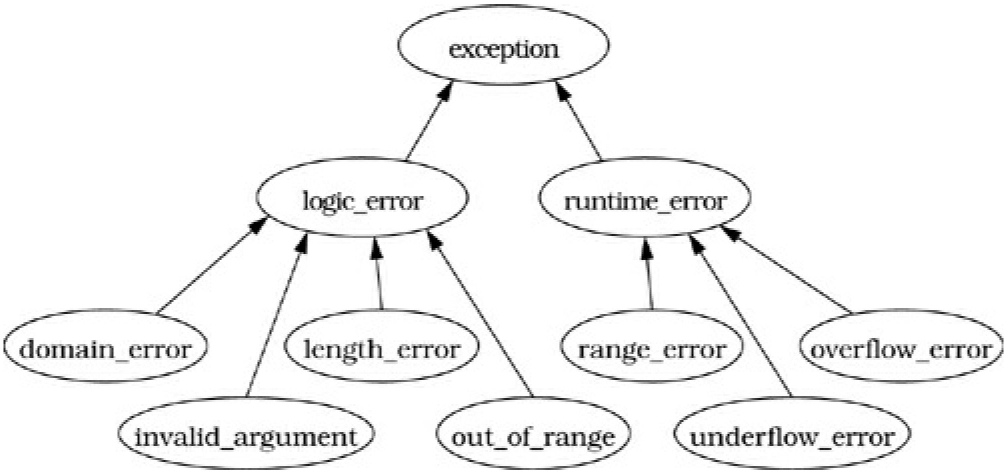
There is nothing surprising here. The language allows implicit conversion from `int` to `double`, so in the call to `sqrt`, `i` is silently converted to a `double`, and the result of `sqrt` corresponds to that `double`. (See Item 5 for a fuller discussion of implicit type conversions.) In general, such conversions are not applied when matching exceptions to `catch` clauses. In this code,

Click here to view code image

```
void f(int value)
{
  try {
    if (someFunction()) {      // if someFunction() returns
      throw value;             // true, throw an int
    }
    ...
  }
  catch (double d) {           // handle exceptions of
    ...                        // type double here
  }

  ...

}
```

the `int` exception thrown inside the `try` block will never be caught by the `catch` clause that takes a `double`. That clause catches only exceptions that are exactly of type `double`; no type conversions are applied. As a result, if the `int` exception is to be caught, it will have to be by some other (dynamically enclosing) `catch` clause taking an `int` or an `int&` (possibly modified by `const` or `volatile`).

Two kinds of conversions *are* applied when matching exceptions to `catch` clauses. The first is inheritance-based conversions. A `catch` clause for base class exceptions is allowed to handle exceptions of (publicly) derived class types, too. For example, consider the diagnostics portion of the hierarchy of exceptions defined by the standard C++ library:


exception
logic_error
runtime_error
domain_error
length_error
range_error
overflow_error
invalid_argument
out_of_range
underflow_error

A `catch` clause for `runtime_error`s can catch exceptions of type `range_error`, `underflow_error`, and `overflow_error`, too, and a `catch` clause accepting an object of the root class `exception` can catch any kind of exception derived from this hierarchy.

This inheritance-based exception-conversion rule applies to values, references, and pointers in the usual fashion (though Item 13 explains why catching values or pointers is generally a bad idea):

Click here to view code image

```
catch (runtime_error) ...        // can catch errors of
type
catch (runtime_error&) ...       // runtime_error,
catch (const runtime_error&) ... // range_error, or
                                 // overflow_error

catch (runtime_error*) ...       // can catch errors of
type
catch (const runtime_error*) ... // runtime_error*,
                                 // range_error*, or
                                 // overflow_error*
```

The second type of allowed conversion is from a typed to an untyped pointer, so a `catch` clause taking a `const void*` pointer will catch an exception of any pointer type:

Click here to view code image

```
catch (const void*) ...          // catches any exception
                                 // that's a pointer
```

The final difference between passing a parameter and propagating an exception is that `catch` clauses are always tried *in the order of their appearance*. Hence, it is possible for an exception of a (publicly) derived class type to be handled by a `catch` clause for one of its base class types — even when a `catch` clause for the derived class is associated with the same `try` block! For example,

Click here to view code image

```
try {
  ...
}
catch (logic_error& ex) {       // this block will catch
  ...                           // all logic_error
}                               // exceptions, even those
                                // of derived types

catch (invalid_argument& ex) {  // this block can never be
  ...                           // executed, because all
}                               // invalid_argument
                                // exceptions will be caught
                                // by the clause above
```

Contrast this behavior with what happens when you call a virtual function. When you call a virtual function, the function invoked is the one in the class *closest* to the dynamic type of the object invoking the function. You might say that virtual functions employ a "best fit" algorithm, while exception handling follows a "first fit" strategy. Compilers may warn you if a `catch` clause for a derived class comes after one for a base class (some issue an error, because such code used to be illegal in C++), but your best course of action is preemptive: never put a `catch` clause for a base class before a `catch` clause for a derived class. The code above, for example, should be reordered like this:

Click here to view code image

```
try {
  ...
}
catch (invalid_argument& ex) { // handle invalid_argument
  ...                          // exceptions here
}
catch (logic_error& ex) {      // handle all other
  ...                          // logic_errors here
}
```

There are thus three primary ways in which passing an object to a function or using that object to invoke a virtual function differs from throwing the object as an exception. First, exception objects are always copied; when caught by value, they are copied twice. Objects passed to function parameters need not be copied at all. Second, objects thrown as exceptions are subject to fewer forms of type conversion than are objects passed to functions. Finally, `catch` clauses are examined in the order in which they appear in the source code, and the first one that can succeed is selected for execution. When an object is used to invoke a virtual function, the function selected is the one that provides the *best* match for the type of the object, even if it's not the first one listed in the source code.

Item 13: Catch exceptions by reference

When you write a `catch` clause, you must specify how exception objects are to be passed to that clause. You have three choices, just as when specifying how parameters should be passed to functions: by pointer, by value, or by reference.

Let us consider first catch by pointer. In theory, this should be the least inefficient way to implement the invariably slow process of moving an exception from `throw` site to `catch` clause (see Item 15). That's because throw by pointer is the only way of moving exception information without copying an object (see Item 12). For example:

Click here to view code image

```
class exception { ... };      // from the standard C++
                              // library exception
                              // hierarchy (see Item 12)

void someFunction()
```

```
{
  static exception ex;          // exception object

  ...
  throw &ex;                    // throw a pointer to ex

  ...

}

void doSomething()
{
  try {
    someFunction();             // may throw an exception*
  }
  catch (exception *ex) {       // catches the exception*;
    ...                         // no object is copied
  }
}
```

This looks neat and tidy, but it's not quite as well-kept as it appears. For this to work, programmers must define exception objects in a way that guarantees the objects exist after control leaves the functions throwing pointers to them. Global and static objects work fine, but it's easy for programmers to forget the constraint. If they do, they typically end up writing code like this:

Click here to view code image

```
void someFunction()
{
  exception ex;                 // local exception object;
                                // will be destroyed when
                                // this function's scope is
  ...                           // exited

  throw &ex;                    // throw a pointer to an
  ...                           // object that's about to
}                               // be destroyed
```

This is worse than useless, because the `catch` clause handling this exception receives a pointer to an object that no longer exists.

An alternative is to throw a pointer to a new heap object:

Click here to view code image

```
void someFunction()
{
  ...
  throw new exception;    // throw a pointer to a new heap-
  ...                     // based object (and hope that
}                         // operator new - see Item 8 --
                          // doesn't itself throw an
                          // exception!)
```

This avoids the I-just-caught-a-pointer-to-a-destroyed-object problem, but now authors of `catch` clauses confront a nasty question: should they delete the pointer they receive? If the exception object was allocated on the heap, they must, otherwise they suffer a resource leak. If the exception object wasn't allocated on the heap, they mustn't, otherwise they suffer undefined program behavior. What to do?

It's impossible to know. Some clients might pass the address of a global or static object, others might pass the address of an exception on the heap. Catch by pointer thus gives rise to the Hamlet conundrum: to delete or not to delete? It's a question with no good answer. You're best off ducking it.

Furthermore, catch-by-pointer runs contrary to the convention established by the language itself. The four standard exceptions — `bad_alloc` (thrown when `operator new` (see Item 8) can't satisfy a memory request), `bad_cast` (thrown when a `dynamic_cast` to a reference fails; see Item 2), `bad_typeid` (thrown when `typeid` is applied to a dereferenced null pointer), and `bad_exception` (available for unexpected exceptions; see Item 14) — are all objects, not pointers to objects, so you have to catch them by value or by reference, anyway.

Catch-by-value eliminates questions about exception deletion and works with the standard exception types. However, it requires that exception objects be copied *twice* each time they're thrown (see Item 12). It also gives rise to the specter of the *slicing problem*, whereby derived class exception objects caught as base class exceptions have their derivedness "sliced off." Such "sliced" objects *are* base class objects: they lack derived class data members, and when virtual functions are called on them, they resolve to virtual functions of the base class. (Exactly the same thing happens when an object is passed to a function by value.) For example, consider an application employing an exception class hierarchy that extends the standard one:

Click here to view code image

```
class exception {                 // as above, this is a
public:                           // standard exception class

  virtual const char * what() const throw();
                                  // returns a brief descrip.
  ...                             // of the exception (see
                                  // Item 14 for info about
};                                // the "throw()" at the
                                  // end of the declaration)

class runtime_error:              // also from the standard
  public exception { ... };       // C++ exception hierarchy

class Validation_error:           // this is a class added by
  public runtime_error {          // a client
public:
  virtual const char * what() const throw();
                                  // this is a redefinition
  ...                             // of the function declared
};                                // in class exception above
void someFunction()               // may throw a validation
{                                 // exception
  ...

  if (a validation test fails) {
    throw Validation_error();
  }

  ...

}

void doSomething()
{
  try {
    someFunction();               // may throw a validation
  }                               // exception

  catch (exception ex) {          // catches all exceptions
                                  // in or derived from
                                  // the standard hierarchy

    cerr << ex.what();            // calls exception::what(),
    ...                           // never
  }                               // Validation_error::what()
}
```

The version of `what` that is called is that of the base class, even though the thrown exception is of type `Validation_error` and `Validation_error` redefines that virtual function. This kind of slicing behavior is almost never what you want.

That leaves only catch-by-reference. Catch-by-reference suffers from none of the problems we have discussed. Unlike catch-by-pointer, the question of object deletion fails to arise, and there is no difficulty in catching the standard exception types. Unlike catch-by-value, there is no slicing problem, and exception objects are copied only once.

If we rewrite the last example using catch-by-reference, it looks like this:

Click here to view code image

```
void someFunction()               // nothing changes in this
{                                 // function
  ...

  if (a validation test fails) {
    throw Validation_error();
  }

  ...

}
void doSomething()
{
  try {
    someFunction();              // no change here
  }
  catch (exception& ex) {        // here we catch by reference
                                 // instead of by value

    cerr << ex.what();           // now calls
                                 // Validation_error::what(),
    ...                          // not exception::what()
  }
}
```

There is no change at the `throw` site, and the only change in the `catch` clause is the addition of an ampersand. This tiny modification makes a big difference, however, because virtual functions in the `catch` block now work as we expect:

functions in `Validation_error` are invoked if they redefine those in `exception`. Of course, if there is no need to modify the exception object in the handler, you'd catch not just by reference, but by reference to `const`.

What a happy confluence of events! If you catch by reference, you sidestep questions about object deletion that leave you damned if you do and damned if you don't; you avoid slicing exception objects; you retain the ability to catch standard exceptions; and you limit the number of times exception objects need to be copied. So what are you waiting for? Catch exceptions by reference!

Item 14: Use exception specifications judiciously

There's no denying it: exception specifications have appeal. They make code easier to understand, because they explicitly state what exceptions a function may throw. But they're more than just fancy comments. Compilers are sometimes able to detect inconsistent exception specifications during compilation. Furthermore, if a function throws an exception not listed in its exception specification, that fault is detected at runtime, and the special function `unexpected` is automatically invoked. Both as a documentation aid and as an enforcement mechanism for constraints on exception usage, then, exception specifications seem attractive.

As is often the case, however, beauty is only skin deep. The default behavior for `unexpected` is to call `terminate`, and the default behavior for `terminate` is to call `abort`, so the default behavior for a program with a violated exception specification is to halt. Local variables in active stack frames are not destroyed, because `abort` shuts down program execution without performing such cleanup. A violated exception specification is therefore a cataclysmic thing, something that should almost never happen.

Unfortunately, it's easy to write functions that make this terrible thing occur. Compilers only *partially* check exception usage for consistency with exception specifications. What they do not check for — what the language standard *prohibits* them from rejecting (though they may issue a warning) — is a call to a function that *might* violate the exception specification of the function making the call.

Consider a declaration for a function `f1` that has no exception specification.

Such a function may throw any kind of exception:

```
extern void f1();             // might throw anything
```

Now consider a function `f2` that claims, through its exception specification, it will throw only exceptions of type `int`:

```
void f2() throw(int);
```

It is perfectly legal C++ for `f2` to call `f1`, even though `f1` might throw an exception that would violate `f2`'s exception specification:

Click here to view code image

```
void f2() throw(int)
{
  ...
  f1();                // legal even though f1 might throw
                       // something besides an int
  ...
}
```

This kind of flexibility is essential if new code with exception specifications is to be integrated with older code lacking such specifications.

Because your compilers are content to let you call functions whose exception specifications are inconsistent with those of the routine containing the calls, and because such calls might result in your program's execution being terminated, it's important to write your software in such a way that these kinds of inconsistencies are minimized. A good way to start is to avoid putting exception specifications on templates that take type arguments. Consider this template, which certainly looks as if it couldn't throw any exceptions:

Click here to view code image

```
// a poorly designed template wrt exception specifications
template<class T>
bool operator==(const T& lhs, const T& rhs) throw()
{
  return &lhs == &rhs;
}
```

This template defines an `operator==` function for all types. For any pair of objects of the same type, it returns `true` if the objects have the same address, otherwise it returns `false`.

This template contains an exception specification stating that the functions generated from the template will throw no exceptions. But that's not necessarily true, because it's possible that `operator&` (the address-of operator) has been overloaded for some types. If it has, `operator&` may throw an exception when called from inside `operator==`. If it does, our exception specification is violated, and off to `unexpected` we go.

This is a specific example of a more general problem, namely, that there is no way to know *anything* about the exceptions thrown by a template's type parameters. We can almost never provide a meaningful exception specification for a template, because templates almost invariably use their type parameter in some way. The conclusion? Templates and exception specifications don't mix.

A second technique you can use to avoid calls to `unexpected` is to omit exception specifications on functions making calls to functions that themselves lack exception specifications. This is simple common sense, but there is one case that is easy to forget. That's when allowing users to register callback functions:

Click here to view code image

```
// Function pointer type for a window system callback
// when a window system event occurs
typedef void (*CallBackPtr)(int eventXLocation,
                            int eventYLocation,
                            void *dataToPassBack);

// Window system class for holding onto callback
// functions registered by window system clients
class CallBack {
public:
  CallBack(CallBackPtr fPtr, void *dataToPassBack)
  : func(fPtr), data(dataToPassBack) {}

  void makeCallBack(int eventXLocation,
                    int eventYLocation) const throw();

private:
  CallBackPtr func;             // function to call when
                                // callback is made

  void *data;                   // data to pass to callback
};                              // function

// To implement the callback, we call the registered func-
// tion with event's coordinates and the registered data
void CallBack::makeCallBack(int eventXLocation,
                            int eventYLocation) const
throw()
{
  func(eventXLocation, eventYLocation, data);
}
```

Here the call to `func` in `makeCallBack` runs the risk of a violated exception specification, because there is no way of knowing what exceptions `func` might throw.

This problem can be eliminated by tightening the exception specification in the `CallBackPtr` typedef:†

Click here to view code image

```
typedef void (*CallBackPtr)(int eventXLocation,
                            int eventYLocation,
                            void *dataToPassBack) throw();
```

Given this typedef, it is now an error to register a `callback` function that fails to guarantee it throws nothing:

Click here to view code image

```
// a callback function without an exception specification
void callBackFcn1(int eventXLocation, int eventYLocation,
                  void *dataToPassBack);

void *callBackData;

...

CallBack c1(callBackFcn1, callBackData);
                               // error! callBackFcn1
                               // might throw an exception

// a callback function with an exception specification
void callBackFcn2(int eventXLocation,
                  int eventYLocation,
                  void *dataToPassBack) throw();
```

```
CallBack c2(callBackFcn2, callBackData);
                               // okay, callBackFcn2 has a
                               // conforming ex. spec.
```

This checking of exception specifications when passing function pointers is a relatively recent addition to the language, so don't be surprised if your compilers don't yet support it. If they don't, it's up to you to ensure you don't make this kind of mistake.

A third technique you can use to avoid calls to `unexpected` is to handle exceptions "the system" may throw. Of these exceptions, the most common is `bad_alloc`, which is thrown by `operator new` and `operator new[]` when a memory allocation fails (see Item 8). If you use the `new` operator (again, see Item 8) in any function, you must be prepared for the possibility that the function will encounter a `bad_alloc` exception.

Now, an ounce of prevention may be better than a pound of cure, but sometimes prevention is hard and cure is easy. That is, sometimes it's easier to cope with unexpected exceptions directly than to prevent them from arising in the first place. If, for example, you're writing software that uses exception specifications rigorously, but you're forced to call functions in libraries that don't use exception specifications, it's impractical to prevent unexpected exceptions from arising, because that would require changing the code in the libraries.

If preventing unexpected exceptions isn't practical, you can exploit the fact that C++ allows you to replace unexpected exceptions with exceptions of a different type. For example, suppose you'd like all unexpected exceptions to be replaced by `UnexpectedException` objects. You can set it up like this,

Click here to view code image

```
class UnexpectedException {};   // all unexpected exception
                                // objects will be replaced
                                // by objects of this type

void convertUnexpected()        // function to call if
{                               // an unexpected exception
  throw UnexpectedException();  // is thrown
}
```

and make it happen by replacing the default `unexpected` function with `convertUnexpected`:

```
set_unexpected(convertUnexpected);
```

Once you've done this, any unexpected exception results in `convertUnexpected` being called. The unexpected exception is then replaced by a new exception of type `UnexpectedException`. Provided the exception specification that was violated includes `UnexpectedException`, exception propagation will then continue as if the exception specification had always been satisfied. (If the exception specification does not include `UnexpectedException`, `terminate` will be called, just as if you had never replaced `unexpected`.)

Another way to translate unexpected exceptions into a well known type is to rely on the fact that if the `unexpected` function's replacement rethrows the current exception, that exception will be replaced by a new exception of the standard type `bad_exception`. Here's how you'd arrange for that to happen:

Click here to view code image

```
void convertUnexpected()        // function to call if
{                               // an unexpected exception
  throw;                        // is thrown; just rethrow
}                               // the current exception

set_unexpected(convertUnexpected);
                                // install convertUnexpected
                                // as the unexpected
                                // replacement
```

If you do this and you include `bad_exception` (or its base class, the standard class `exception`) in all your exception specifications, you'll never have to worry about your program halting if an unexpected exception is encountered. Instead, any wayward exception will be replaced by a `bad_exception`, and that exception will be propagated in the stead of the original one.

By now you understand that exception specifications can be a lot of trouble. Compilers perform only partial checks for their consistent usage, they're problematic in templates, they're easy to violate inadvertently, and, by default, they lead to abrupt program termination when they're violated. Exception specifications have another drawback, too, and that's that they result in `unexpected` being invoked even when a higher-level caller is prepared to cope with the exception that's arisen. For example, consider this code, which is taken almost verbatim from Item 11:

Click here to view code image

```
class Session {                  // for modeling online
public:                          // sessions
  ~Session();
  ...

private:
  static void logDestruction(Session *objAddr) throw();
};

Session::~Session()
{
  try {
    logDestruction(this);
  }
  catch (...) {}
}
```

The `Session` destructor calls `logDestruction` to record the fact that a `Session` object is being destroyed, but it explicitly catches any exceptions that might be thrown by `logDestruction`. However, `logDestruction` comes with an exception specification asserting that it throws no exceptions. Now, suppose some function called by `logDestruction` throws an exception that `logDestruction` fails to catch. This isn't supposed to happen, but as we've seen, it isn't difficult to write code that leads to the violation of exception specifications. When this unanticipated exception propagates through `logDestruction`, `unexpected` will be called, and, by default, that will result in termination of the program. This is correct behavior, to be sure, but is it the behavior the author of `Session`'s destructor wanted? That author took pains to handle *all possible* exceptions, so it seems almost unfair to halt the program without giving `Session`'s destructor's `catch` block a chance to work. If `logDestruction` had no exception specification, this I'm-willing-to-catch-it-if-you'll-just-give-me-a-chance scenario would never arise. (One way to prevent it is to replace `unexpected` as described above.)

It's important to keep a balanced view of exception specifications. They provide excellent documentation on the kinds of exceptions a function is expected to throw, and for situations in which violating an exception specification is so dire as to justify immediate program termination, they offer that behavior by default. At the same time, they are only partly checked by compilers and they are easy to violate inadvertently. Furthermore, they can prevent high-level exception handlers from dealing with unexpected exceptions, even when they know how to. That being the case, exception specifications are a tool to be applied judiciously. Before adding them to your functions, consider whether the behavior they impart to your software is really the behavior you want.

Item 15: Understand the costs of exception handling

To handle exceptions at runtime, programs must do a fair amount of bookkeeping. At each point during execution, they must be able to identify the objects that require destruction if an exception is thrown; they must make note of each entry to and exit from a `try` block; and for each `try` block, they must keep track of the associated `catch` clauses and the types of exceptions those clauses can handle. This bookkeeping is not free. Nor are the runtime comparisons necessary to ensure that exception specifications are satisfied. Nor is the work expended to destroy the appropriate objects and find the correct `catch` clause when an exception is thrown. No, exception handling has costs, and you pay at least some of them even if you never use the keywords `try`, `throw`, or `catch`.

Let us begin with the things you pay for even if you never use any exception-handling features. You pay for the space used by the data structures needed to keep track of which objects are fully constructed (see Item 10), and you pay for the time needed to keep these data structures up to date. These costs are typically quite modest. Nevertheless, programs compiled without support for exceptions are typically both faster and smaller than their counterparts compiled with support for exceptions.

In theory, you don't have a choice about these costs: exceptions are part of C++, compilers have to support them, and that's that. You can't even expect compiler vendors to eliminate the costs if you use no exception-handling features, because programs are typically composed of multiple independently generated object files, and just because one object file doesn't do anything with exceptions doesn't mean others don't. Furthermore, even if none of the object files linked to form an executable use exceptions, what about the libraries they're linked with? If *any part* of a program uses exceptions, the rest of the program must support them, too. Otherwise it may not be possible to provide correct exception-handling behavior at runtime.

That's the theory. In practice, most vendors who support exception handling allow you to control whether support for exceptions is included in the code they generate. If you know that no part of your program uses `try`, `throw`, or `catch`, and you also know that no library with which you'll link uses `try`, `throw`, or `catch`, you might as well compile without exception-handling support and save yourself the size and speed penalty you'd otherwise probably be assessed for a feature you're not using. As time goes on and libraries employing exceptions become more common, this strategy will become less tenable, but given the current state of C++ software development, compiling without support for exceptions is a reasonable performance optimization if you have already decided not to use exceptions. It may also be an attractive optimization for libraries that eschew exceptions, provided they can guarantee that exceptions thrown from client code never propagate into the library. This is a difficult guarantee to make, as it precludes client redefinitions of library-declared virtual functions; it also rules out client-defined callback functions.

A second cost of exception-handling arises from `try` blocks, and you pay it whenever you use one, i.e., whenever you decide you want to be able to catch exceptions. Different compilers implement `try` blocks in different ways, so the cost varies from compiler to compiler. As a rough estimate, expect your overall code size to increase by 5-10% and your runtime to go up by a similar amount if you use `try` blocks. This assumes no exceptions are thrown; what we're discussing here is just the cost of *having* `try` blocks in your programs. To minimize this cost, you should avoid unnecessary `try` blocks.

Compilers tend to generate code for exception specifications much as they do for `try` blocks, so an exception specification generally incurs about the same cost as a `try` block. Excuse me? You say you thought exception specifications were just specifications, you didn't think they generated code? Well, now you have something new to think about.

Which brings us to the heart of the matter, the cost of throwing an exception. In truth, this shouldn't be much of a concern, because exceptions should be rare. After all, they indicate the occurrence of events that are *exceptional*. The 80-20 rule (see Item 16) tells us that such events should almost never have much impact on a program's overall performance. Nevertheless, I know you're curious about just how big a hit you'll take if you throw an exception, and the answer is it's probably a big one. Compared to a normal function return, returning from a function by throwing an exception may be as much as *three orders of magnitude* slower. That's quite a hit. But you'll take it only if you throw an exception, and that should be almost never. If, however, you've been thinking of using exceptions to indicate relatively common conditions like the completion of a data structure traversal or the termination of a loop, now would be an excellent time to think again.

But wait. How can I know this stuff? If support for exceptions is a relatively recent addition to most compilers (it is), and if different compilers implement their support in different ways (they do), how can I say that a program's size will generally grow by about 5-10%, its speed will decrease by a similar amount, and it may run orders of magnitude slower if lots of exceptions are thrown? The answer is frightening: a little rumor and a handful of benchmarks (see Item 23). The fact is that most people — including most compiler vendors — have little experience with exceptions, so though we know there are costs associated with them, it is difficult to predict those costs accurately.

The prudent course of action is to be aware of the costs described in this item, but not to take the numbers very seriously. Whatever the cost of exception handling, you don't want to pay any more than you have to. To minimize your exception-related costs, compile without support for exceptions when that is feasible; limit your use of `try` blocks and exception specifications to those locations where you honestly need them; and throw exceptions only under conditions that are truly exceptional. If you still have performance problems, profile your software (see Item 16) to determine if exception support is a contributing factor. If it is, consider switching to different compilers, ones that provide more efficient implementations of C++'s exception-handling features.

Efficiency

I harbor a suspicion that someone has performed secret Pavlovian experiments on C++ software developers. How else can one explain the fact that when the word "efficiency" is mentioned, scores of programmers start to drool?

In fact, efficiency is no laughing matter. Programs that are too big or too slow fail to find acceptance, no matter how compelling their merits. This is perhaps as it should be. Software is supposed to help us do things better, and it's difficult to argue that slower is better, that demanding 32 megabytes of memory is better than requiring a mere 16, that chewing up 100 megabytes of disk space is better than swallowing only 50. Furthermore, though some programs take longer and use more memory because they perform more ambitious computations, too many programs can blame their sorry pace and bloated footprint on nothing more than bad design and slipshod programming.

Writing efficient programs in C++ starts with the recognition that C++ may well have nothing to do with any performance problems you've been having. If you want to write an efficient C++ program, you must first be able to write an efficient *program*. Too many developers overlook this simple truth. Yes, loops may be unrolled by hand and multiplications may be replaced by shift operations, but such micro-tuning leads nowhere if the higher-level algorithms you employ are inherently inefficient. Do you use quadratic algorithms when linear ones are available? Do you compute the same value over and over? Do you squander opportunities to reduce the average cost of expensive operations? If so, you can hardly be surprised if your programs are described like second-rate tourist attractions: worth a look, but only if you've got some extra time.

The material in this chapter attacks the topic of efficiency from two angles. The first is language-independent, focusing on things you can do in any programming language. C++ provides a particularly appealing implementation medium for these ideas, because its strong support for encapsulation makes it possible to replace inefficient class implementations with better algorithms and data structures that support the same interface.

The second focus is on C++ itself. High-performance algorithms and data structures are great, but sloppy implementation practices can reduce their effectiveness considerably. The most insidious mistake is both simple to make and hard to recognize: creating and destroying too many objects. Superfluous object constructions and destructions act like a hemorrhage on your program's performance, with precious clock-ticks bleeding away each time an unnecessary object is created and destroyed. This problem is so pervasive in C++ programs, I devote four separate items to describing where these objects come from and how you can eliminate them without compromising the correctness of your code.

Programs don't get big and slow only by creating too many objects. Other potholes on the road to high performance include library selection and implementations of language features. In the items that follow, I address these issues, too.

After reading the material in this chapter, you'll be familiar with several principles that can improve the performance of virtually any program you write, you'll know exactly how to prevent unnecessary objects from creeping into your software, and you'll have a keener awareness of how your compilers behave when generating executables.

It's been said that forewarned is forearmed. If so, think of the information that follows as preparation for battle.

Item 16: Remember the 80-20 rule

The 80-20 rule states that 80 percent of a program's resources are used by about 20 percent of the code: 80 percent of the runtime is spent in approximately 20 percent of the code; 80 percent of the memory is used by some 20 percent of the code; 80 percent of the disk accesses are performed for about 20 percent of the code; 80 percent of the maintenance effort is devoted to around 20 percent of the code. The rule has been repeatedly verified through examinations of countless machines, operating systems, and applications. The 80-20 rule is more than just a catchy phrase; it's a guideline about system performance that has both wide applicability and a solid empirical basis.

When considering the 80-20 rule, it's important not to get too hung up on numbers. Some people favor the more stringent 90-10 rule, and there's experimental evidence to back that, too. Whatever the precise numbers, the fundamental point is this: the overall performance of your software is almost always determined by a small part of its constituent code.

As a programmer striving to maximize your software's performance, the 80-20 rule both simplifies and complicates your life. On one hand, the 80-20 rule implies that most of the time you can produce code whose performance is, frankly, rather mediocre, because 80 percent of the time its efficiency doesn't affect the overall performance of the system you're working on. That may not do much for your ego, but it should reduce your stress level a little. On the other hand, the rule implies that if your software has a performance problem, you've got a tough job ahead of you, because you not only have to locate the small pockets of code that are causing the problem, you have to find ways to increase their performance dramatically. Of these tasks, the more troublesome is generally locating the bottlenecks. There are two fundamentally different ways to approach the matter: the way most people do it and the right way.

The way most people locate bottlenecks is to guess. Using experience, intuition, tarot cards and Ouija boards, rumors or worse, developer after developer solemnly proclaims that a program's efficiency problems can be traced to network delays, improperly tuned memory allocators, compilers that don't optimize aggressively enough, or some bonehead manager's refusal to permit assembly language for crucial inner loops. Such assessments are generally delivered with a condescending sneer, and usually both the sneerers and their prognostications are flat-out wrong.

Most programmers have lousy intuition about the performance characteristics of their programs, because program performance characteristics tend to be highly unintuitive. As a result, untold effort is poured into improving the efficiency of parts of programs that will never have a noticeable effect on their overall behavior. For example, fancy algorithms and data structures that minimize computation may be added to a program, but it's all for naught if the program is I/O-bound. Souped-up I/O libraries (see Item 23) may be substituted for the ones shipped with compilers, but there's not much point if the programs using them are CPU-bound.

That being the case, what do you do if you're faced with a slow program or one that uses too much memory? The 80-20 rule means that improving random parts of the program is unlikely to help very much. The fact that programs tend to have unintuitive performance characteristics means that trying to guess the causes of performance bottlenecks is unlikely to be much better than just improving random parts of your program. What, then, *will* work?

What will work is to empirically identify the 20 percent of your program that is causing you heartache, and the way to identify that horrid 20 percent is to use a program profiler. Not just any profiler will do, however. You want one that *directly* measures the resources you are interested in. For example, if your program is too slow, you want a profiler that tells you how much *time* is being spent in different parts of the program. That way you can focus on those places where a significant improvement in local efficiency will also yield a significant improvement in overall efficiency.

Profilers that tell you how many times each statement is executed or how many times each function is called are of limited utility. From a performance point of view, you do not care how many times a statement is executed or a function is called. It is, after all, rather rare to encounter a user of a program or a client of a library who complains that too many statements are being executed or too many functions are being called. If your software is fast enough, nobody cares how many statements are executed, and if it's too slow, nobody cares how few. All they care about is that they hate to wait, and if your program is making them do it, they hate you, too.

Still, knowing how often statements are executed or functions are called can sometimes yield insight into what your software is doing. If, for example, you think you're creating about a hundred objects of a particular type, it would certainly be worthwhile to discover that you're calling constructors in that class thousands of times. Furthermore, statement and function call counts can indirectly help you understand facets of your software's behavior you can't directly measure. If you have no direct way of measuring dynamic memory usage, for example, it may be helpful to know at least how often memory allocation and deallocation functions (e.g., operators `new`, `new[]`, `delete`, and `delete[]` — see Item 8) are called.

Of course, even the best of profilers is hostage to the data it's given to process. If you profile your program while it's processing unrepresentative input data, you're in no position to complain if the profiler leads you to fine-tune parts of your software — the parts making up some 80 percent of it — that have no bearing on its usual performance. Remember that a profiler can only tell you how a program behaved on a particular run (or set of runs), so if you profile a program using input data that is unrepresentative, you're going to get back a profile that is equally unrepresentative. That, in turn, is likely to lead to you to optimize your software's behavior for uncommon uses, and the overall impact on common uses may even be negative.

The best way to guard against these kinds of pathological results is to profile your software using as many data sets as possible. Moreover, you must ensure that each data set is representative of how the software is used by its clients (or at least its most important clients). It is usually easy to acquire representative data sets, because many clients are happy to let you use their data when profiling. After all, you'll then be tuning your software to meet their needs, and that can only be good for both of you.

Item 17: Consider using lazy evaluation

From the perspective of efficiency, the best computations are those you never perform at all. That's fine, but if you don't need to do something, why would you put code in your program to do it in the first place? And if you do need to do something, how can you possibly avoid executing the code that does it?

The key is to be lazy.

Remember when you were a child and your parents told you to clean your room? If you were anything like me, you'd say "Okay," then promptly go back to what you were doing. You would *not* clean your room. In fact, cleaning your room would be the last thing on your mind — *until* you heard your parents coming down the hall to confirm that your room had, in fact, been cleaned. Then you'd sprint to your room and get to work as fast as you possibly could. If you were lucky, your parents would never check, and you'd avoid all the work cleaning your room normally entails.

It turns out that the same delay tactics that work for a five year old work for a C++ programmer. In Computer Science, however, we dignify such procrastination with the name *lazy evaluation*. When you employ lazy evaluation, you write your classes in such a way that they defer computations until the *results* of those computations are required. If the results are never required, the computations are never performed, and neither your software's clients nor your parents are any the wiser.

Perhaps you're wondering exactly what I'm talking about. Perhaps an example would help. Well, lazy evaluation is applicable in an enormous variety of application areas, so I'll describe four.

Reference Counting

Consider this code:

Click here to view code image

```
class String { ... };          // a string class (the
standard
                               // string type may be

implemented
                               // as described below, but

it
                               // doesn't have to be)

String s1 = "Hello";

String s2 = s1;                // call String copy ctor
```

A common implementation for the `String` copy constructor would result in `s1` and `s2` each having its own copy of `"Hello"` after `s2` is initialized with `s1`. Such a copy constructor would incur a relatively large expense, because it would have to make a copy of `s1`'s value to give to `s2`, and that would typically entail allocating heap memory via the `new` operator (see Item 8) and calling `strcpy` to copy the data in `s1` into the memory allocated by `s2`. This is *eager evaluation*: making a copy of `s1` and putting it into `s2` just because the `String` copy constructor was *called*. At this point, however, there has been no real *need* for `s2` to have a copy of the value, because `s2` hasn't been used yet.

The lazy approach is a lot less work. Instead of giving `s2` a copy of `s1`'s value, we have `s2` *share* `s1`'s value. All we have to do is a little bookkeeping so we know who's sharing what, and in return we save the cost of a call to `new` and the expense of copying anything. The fact that `s1` and `s2` are sharing a data structure is transparent to clients, and it certainly makes no difference in statements like the following, because they only read values, they don't write them:

[Click here to view code image](#)

```
cout << s1;                   // read s1's value

cout << s1 + s2;              // read s1's and s2's values
```

In fact, the only time the sharing of values makes a difference is when one or the other string is *modified*; then it's important that only one string be changed, not both. In this statement,

```
s2.convertToUpperCase();
```

it's crucial that only `s2`'s value be changed, not `s1`'s also.

To handle statements like this, we have to implement `String`'s `convertToUpperCase` function so that it makes a copy of `s2`'s value and makes that value private to `s2` before modifying it. Inside `convertToUpperCase`, we can be lazy no longer: we have to make a copy of `s2`'s (shared) value for `s2`'s private use. On the other hand, if `s2` is never modified, we never have to make a private copy of its value. It can continue to share a value as long as it exists. If we're lucky, `s2` will never be modified, in which case we'll never have to expend the effort to give it its own value.

The details on making this kind of value sharing work (including all the code) are provided in [Item 29](#), but the idea is lazy evaluation: don't bother to make a copy of something until you really need one. Instead, be lazy — use someone else's copy as long as you can get away with it. In some application areas, you can often get away with it forever.

Distinguishing Reads from Writes

Pursuing the example of reference-counting strings a bit further, we come upon a second way in which lazy evaluation can help us. Consider this code:

[Click here to view code image](#)

```
String s = "Homer's Iliad";    // Assume s is a
                               // reference-counted string
...

cout << s[3];                // call operator[] to read s[3]
s[3] = 'x';                  // call operator[] to write s[3]
```

The first call to `operator[]` is to read part of a string, but the second call is to perform a write. We'd like to be able to distinguish the read call from the write, because reading a reference-counted string is cheap, but writing to such a string may require splitting off a new copy of the string's value prior to the write.

This puts us in a difficult implementation position. To achieve what we want, we need to do different things inside `operator[]` (depending on whether it's being called to perform a read or a write). How can we determine whether `operator[]` has been called in a read or a write context? The brutal truth is that we can't. By using lazy evaluation and proxy classes as described in Item 30, however, we can defer the decision on whether to take read actions or write actions until we can determine which is correct.

Lazy Fetching

As a third example of lazy evaluation, imagine you've got a program that uses large objects containing many constituent fields. Such objects must persist across program runs, so they're stored in a database. Each object has a unique object identifier that can be used to retrieve the object from the database:

Click here to view code image

```
class LargeObject {                // large persistent
objects
public:
  LargeObject(ObjectID id);        // restore object from
disk

  const string& field1() const;    // value of field 1
  int field2() const;              // value of field 2
  double field3() const;           // ...
  const string& field4() const;
  const string& field5() const;
  ...

};
```

Now consider the cost of restoring a `LargeObject` from disk:

Click here to view code image

```
void restoreAndProcessObject(ObjectID id)
{
  LargeObject object(id);        // restore object
```

```
    ...

  }
```

Because `LargeObject` instances are big, getting all the data for such an object might be a costly database operation, especially if the data must be retrieved from a remote database and pushed across a network. In some cases, the cost of reading all that data would be unnecessary. For example, consider this kind of application:

Click here to view code image

```
void restoreAndProcessObject(ObjectID id)
{
  LargeObject object(id);

  if (object.field2() == 0) {
    cout << "Object " << id << ": null field2.\n";
  }
}
```

Here only the value of `field2` is required, so any effort spent setting up the other fields is wasted.

The lazy approach to this problem is to read no data from disk when a `LargeObject` object is created. Instead, only the "shell" of an object is created, and data is retrieved from the database only when that particular data is needed inside the object. Here's one way to implement this kind of "demand-paged" object initialization:

Click here to view code image

```
class LargeObject {
public:
  LargeObject(ObjectID id);

  const string& field1() const;
  int field2() const;
  double field3() const;
  const string& field4() const;
  ...

private:
  ObjectID oid;
```

```
    mutable string *field1Value;    // see below for a
    mutable int *field2Value;       // discussion of "mutable"
    mutable double *field3Value;
    mutable string *field4Value;
    ...

  };

  LargeObject::LargeObject(ObjectID id)
  : oid(id), field1Value(0), field2Value(0), field3Value(0),
  ...
  {}

  const string& LargeObject::field1() const
  {
    if (field1Value == 0) {
      read the data for field 1 from the database and make
      field1Value point to it;
    }

    return *field1Value;
  }
```

Each field in the object is represented as a pointer to the necessary data, and the `LargeObject` constructor initializes each pointer to null. Such null pointers signify fields that have not yet been read from the database. Each `LargeObject` member function must check the state of a field's pointer before accessing the data it points to. If the pointer is null, the corresponding data must be read from the database before performing any operations on that data.

When implementing lazy fetching, you must confront the problem that null pointers may need to be initialized to point to real data from inside any member function, including `const` member functions like `field1`. However, compilers get cranky when you try to modify data members inside `const` member functions, so you've got to find a way to say, "It's okay, I know what I'm doing." The best way to say that is to declare the pointer fields `mutable`, which means they can be modified inside any member function, even inside `const` member functions. That's why the fields inside `LargeObject` above are declared `mutable`.

The `mutable` keyword is a relatively recent addition to C++, so it's possible your vendors don't yet support it. If not, you'll need to find another way to convince your compilers to let you modify data members inside `const` member functions. One workable strategy is the "fake `this`" approach, whereby you create a pointer-to-non-`const` that points to the same object as `this` does. When you want to modify a data member, you access it through the "fake `this`" pointer:

Click here to view code image

```
class LargeObject {
public:
  const string& field1() const;    // unchanged
  ...

private:
  string *field1Value;             // not declared mutable
  ...                              // so that older
};                                 // compilers will accept
it
const string& LargeObject::field1() const
{
  // declare a pointer, fakeThis, that points where this
  // does, but where the constness of the object has been
  // cast away
  LargeObject * const fakeThis =
    const_cast<LargeObject* const>(this);

  if (field1Value == 0) {
    fakeThis->field1Value =        // this assignment is OK,
      the appropriate data         // because what fakeThis
      from the database;           // points to isn't const
  }

  return *field1Value;
}
```

This function employs a `const_cast` (see Item 2) to cast away the `const`ness of `*this`. If your compilers don't support `const_cast`, you can use an old C-style cast:

Click here to view code image

```
// Use of old-style cast to help emulate mutable
const string& LargeObject::field1() const
{
  LargeObject * const fakeThis = (LargeObject* const)this;

  ...                             // as above
}
```

Look again at the pointers inside `LargeObject`. Let's face it, it's tedious and error-prone to have to initialize all those pointers to null, then test each one before use. Fortunately, such drudgery can be automated through the use of *smart pointers*, which you can read about in Item 28. If you use smart pointers inside `LargeObject`, you'll also find you no longer need to declare the pointers `mutable`. Alas, it's only a temporary respite, because you'll wind up needing `mutable` once you sit down to implement the smart pointer classes. Think of it as conservation of inconvenience.

Lazy Expression Evaluation

A final example of lazy evaluation comes from numerical applications. Consider this code:

Click here to view code image

```
template<class T>
class Matrix { ... };           // for homogeneous matrices

Matrix<int> m1(1000, 1000);     // a 1000 by 1000 matrix
Matrix<int> m2(1000, 1000);     // ditto

...

Matrix<int> m3 = m1 + m2;       // add m1 and m2
```

The usual implementation of `operator+` would use eager evaluation; in this case it would compute and return the sum of `m1` and `m2`. That's a fair amount of computation (1,000,000 additions), and of course there's the cost of allocating the memory to hold all those values, too.

The lazy evaluation strategy says that's *way* too much work, so it doesn't do it. Instead, it sets up a data structure inside `m3` that indicates that `m3`'s value is the sum of `m1` and `m2`. Such a data structure might consist of nothing more than a pointer to each of `m1` and `m2`, plus an enum indicating that the operation on them is addition. Clearly, it's going to be faster to set up this data structure than to add `m1` and `m2`, and it's going to use a lot less memory, too.

Suppose that later in the program, before `m3` has been used, this code is executed:

[Click here to view code image](#)

```
Matrix<int> m4(1000, 1000);

...                              // give m4 some values

m3 = m4 * m1;
```

Now we can forget all about `m3` being the sum of `m1` and `m2` (and thereby save the cost of the computation), and in its place we can start remembering that `m3` is the product of `m4` and `m1`. Needless to say, we don't perform the multiplication. Why bother? We're lazy, remember?

This example looks contrived, because no good programmer would write a program that computed the sum of two matrices and failed to use it, but it's not as contrived as it seems. No good programmer would deliberately compute a value that's not needed, but during maintenance, it's not uncommon for a programmer to modify the paths through a program in such a way that a formerly useful computation becomes unnecessary. The likelihood of that happening is reduced by defining objects immediately prior to use, but it's still a problem that occurs from time to time.

Nevertheless, if that were the only time lazy evaluation paid off, it would hardly be worth the trouble. A more common scenario is that we need only *part* of a computation. For example, suppose we use `m3` as follows after initializing it to the sum of `m1` and `m2`:

```
cout << m3[4];                   // print the 4th row of m3
```

Clearly we can be completely lazy no longer — we've got to compute the values in the fourth row of `m3`. But let's not be overly ambitious, either. There's no reason we have to compute any *more* than the fourth row of `m3`; the remainder of `m3` can remain uncomputed until it's actually needed. With luck, it never will be.

How likely are we to be lucky? Experience in the domain of matrix computations suggests the odds are in our favor. In fact, lazy evaluation lies behind the wonder that is APL. APL was developed in the 1960s for interactive use by people who needed to perform matrix-based calculations. Running on computers that had less computational horsepower than the chips now found in high-end microwave ovens, APL was seemingly able to add, multiply, and even divide large matrices instantly! Its trick was lazy evaluation. The trick was usually effective, because APL users typically added, multiplied, or divided matrices not because they needed the entire resulting matrix, but only because they needed a small part of it. APL employed lazy evaluation to defer its computations until it knew exactly what part of a result matrix was needed, then it computed only that part. In practice, this allowed users to perform computationally intensive tasks *interactively* in an environment where the underlying machine was hopelessly inadequate for an implementation employing eager evaluation. Machines are faster today, but data sets are bigger and users less patient, so many contemporary matrix libraries continue to take advantage of lazy evaluation.

To be fair, laziness sometimes fails to pay off. If `m3` is used in this way,

```
cout << m3;                        // print out all of m3
```

the jig is up and we've got to compute a complete value for `m3`. Similarly, if one of the matrices on which `m3` is dependent is about to be modified, we have to take immediate action:

Click here to view code image

```
m3 = m1 + m2;                      // remember that m3 is the
                                   // sum of m1 and m2

m1 = m4;                           // now m3 is the sum of m2
                                   // and the OLD value of m1!
```

Here we've got to do something to ensure that the assignment to `m1` doesn't change `m3`. Inside the `Matrix<int>` assignment operator, we might compute `m3`'s value prior to changing `m1` or we might make a copy of the old value of `m1` and make `m3` dependent on that, but we have to do *something* to guarantee that `m3` has the value it's supposed to have after `m1` has been the target of an assignment. Other functions that might modify a matrix must be handled in a similar fashion.

Because of the need to store dependencies between values; to maintain data structures that can store values, dependencies, or a combination of the two; and to overload operators like assignment, copying, and addition, lazy evaluation in a numerical domain is a lot of work. On the other hand, it often ends up saving significant amounts of time and space during program runs, and in many applications, that's a payoff that easily justifies the significant effort lazy evaluation requires.

Summary

These four examples show that lazy evaluation can be useful in a variety of domains: to avoid unnecessary copying of objects, to distinguish reads from writes using `operator[]`, to avoid unnecessary reads from databases, and to avoid unnecessary numerical computations. Nevertheless, it's not always a good idea. Just as procrastinating on your clean-up chores won't save you any work if your parents always check up on you, lazy evaluation won't save your program any work if all your computations are necessary. Indeed, if all your computations are essential, lazy evaluation may slow you down and increase your use of memory, because, in addition to having to do all the computations you were hoping to avoid, you'll also have to manipulate the fancy data structures needed to make lazy evaluation possible in the first place. Lazy evaluation is only useful when there's a reasonable chance your software will be asked to perform computations that can be avoided.

There's nothing about lazy evaluation that's specific to C++. The technique can be applied in any programming language, and several languages — notably APL, some dialects of Lisp, and virtually all dataflow languages — embrace the idea as a fundamental part of the language. Mainstream programming languages employ eager evaluation, however, and C++ is mainstream. Yet C++ is particularly suitable as a vehicle for user-implemented lazy evaluation, because its support for encapsulation makes it possible to add lazy evaluation to a class without clients of that class knowing it's been done.

Look again at the code fragments used in the above examples, and you can verify that the class interfaces offer no hints about whether eager or lazy evaluation is used by the classes. That means it's possible to implement a class using a straightforward eager evaluation strategy, but then, if your profiling investigations (see Item 16) show that class's implementation is a performance bottleneck, you can replace its implementation with one based on lazy evaluation. The only change your clients will see (after recompilation or relinking) is improved performance. That's the kind of software enhancement clients love, one that can make you downright proud to be lazy.

Item 18: Amortize the cost of expected computations

In Item 17, I extolled the virtues of laziness, of putting things off as long as possible, and I explained how laziness can improve the efficiency of your programs. In this item, I adopt a different stance. Here, laziness has no place. I now encourage you to improve the performance of your software by having it do *more* than it's asked to do. The philosophy of this item might be called *over-eager evaluation*: doing things *before* you're asked to do them.

Consider, for example, a template for classes representing large collections of numeric data:

Click here to view code image

```
template<class NumericalType>
class DataCollection {
public:
  NumericalType min() const;
  NumericalType max() const;
  NumericalType avg() const;
  ...
};
```

Assuming the `min`, `max`, and `avg` functions return the current minimum, maximum, and average values of the collection, there are three ways in which these functions can be implemented. Using eager evaluation, we'd examine all the data in the collection when `min`, `max`, or `avg` was called, and we'd return the appropriate value. Using lazy evaluation, we'd have the functions return data structures that could be used to determine the appropriate value whenever the functions' return values were actually used. Using over-eager evaluation, we'd keep track of the running minimum, maximum, and average values of the collection, so when `min`, `max`, or `avg` was called, we'd be able to return the correct value immediately — no computation would be required. If `min`, `max`, and `avg` were called frequently, we'd be able to amortize the cost of keeping track of the collection's minimum, maximum, and average values over all the calls to those functions, and the amortized cost per call would be lower than with eager or lazy evaluation.

The idea behind over-eager evaluation is that if you expect a computation to be requested frequently, you can lower the average cost per request by designing your data structures to handle the requests especially efficiently.

One of the simplest ways to do this is by caching values that have already been computed and are likely to be needed again. For example, suppose you're writing a program to provide information about employees, and one of the pieces of information you expect to be requested frequently is an employee's cubicle number. Further suppose that employee information is stored in a database, but, for most applications, an employee's cubicle number is irrelevant, so the database is not optimized to find it. To avoid having your specialized application unduly stress the database with repeated lookups of employee cubicle numbers, you could write a `findCubicleNumber` function that caches the cubicle numbers it looks up. Subsequent requests for cubicle numbers that have already been retrieved can then be satisfied by consulting the cache instead of querying the database.

Here's one way to implement `findCubicleNumber`; it uses a `map` object from the Standard Template Library (the "STL" — see Item 35) as a local cache:

Click here to view code image

```
int findCubicleNumber(const string& employeeName)
{
  // define a static map to hold (employee name, cubicle
number)
  // pairs. This map is the local cache.
  typedef map<string, int> CubicleMap;
  static CubicleMap cubes;

  // try to find an entry for employeeName in the cache;
  // the STL iterator "it" will then point to the found
  // entry, if there is one (see Item 35 for details)
  CubicleMap::iterator it = cubes.find(employeeName);

  // "it"'s value will be cubes.end() if no entry was
  // found (this is standard STL behavior). If this is
  // the case, consult the database for the cubicle
  // number, then add it to the cache
  if (it == cubes.end()) {
    int cubicle =
      the result of looking up employeeName's cubicle
      number in the database;

    cubes[employeeName] = cubicle;   // add the pair
                                     // (employeeName,
cubicle)
                                     // to the cache
    return cubicle;
  }
```

```
    else {
      // "it" points to the correct cache entry, which is a
      // (employee name, cubicle number) pair. We want only
      // the second component of this pair, and the member
      // "second" will give it to us
      return (*it).second;
    }
  }
```

Try not to get bogged down in the details of the STL code (which will be clearer after you've read Item 35). Instead, focus on the general strategy embodied by this function. That strategy is to use a local cache to replace comparatively expensive database queries with comparatively inexpensive lookups in an in-memory data structure. Provided we're correct in assuming that cubicle numbers will frequently be requested more than once, the use of a cache in `findCubicleNumber` should reduce the average cost of returning an employee's cubicle number.

(One detail of the code requires explanation. The final statement returns `(*it).second` instead of the more conventional `it->second`. Why? The answer has to do with the conventions followed by the STL. In brief, the iterator `it` is an object, not a pointer, so there is no guarantee that "`->`" can be applied to `it`.[‡] The STL does require that "`.`" and "`*`" be valid for iterators, however, so `(*it).second`, though syntactically clumsy, is guaranteed to work.)

Caching is one way to amortize the cost of anticipated computations. Prefetching is another. You can think of prefetching as the computational equivalent of a discount for buying in bulk. Disk controllers, for example, read entire blocks or sectors of data when they read from disk, even if a program asks for only a small amount of data. That's because it's faster to read a big chunk once than to read two or three small chunks at different times. Furthermore, experience has shown that if data in one place is requested, it's quite common to want nearby data, too. This is the infamous *locality of reference* phenomenon, and systems designers rely on it to justify disk caches, memory caches for both instructions and data, and instruction prefetches.

Excuse me? You say you don't worry about such low-level things as disk controllers or CPU caches? No problem. Prefetching can yield dividends for even one as high-level as you. Imagine, for example, you'd like to implement a template for dynamic arrays, i.e., arrays that start with a size of one and automatically extend themselves so that all nonnegative indices are valid:

Click here to view code image

```
template<class T>                // template for dynamic
class DynArray { ... };          // array-of-T classes

DynArray<double> a;              // at this point, only a[0]
                                 // is a legitimate array
                                 // element

a[22] = 3.5;                     // a is automatically
                                 // extended: valid indices
                                 // are now 0-22

a[32] = 0;                       // a extends itself again;
                                 // now a[0]-a[32] are valid
```

How does a `DynArray` object go about extending itself when it needs to? A straightforward strategy would be to allocate only as much additional memory as needed, something like this:

Click here to view code image

```
template<class T>
T& DynArray<T>::operator[](int index)
{
  if (index < 0) {
    throw an exception;          // negative indices are
  }                              // still invalid

  if (index > the current maximum index value) {
    call new to allocate enough additional memory so that
    index is valid;
  }

  return the indexth element of the array;
}
```

This approach simply calls `new` each time it needs to increase the size of the array, but calls to `new` invoke `operator new` (see Item 8), and calls to `operator new` (and `operator delete`) are usually expensive. That's because they typically result in calls to the underlying operating system, and system calls are generally slower than are in-process function calls. As a result, we'd like to make as few system calls as possible.

An over-eager evaluation strategy employs this reasoning: if we have to increase the size of the array now to accommodate index *i*, the locality of reference principle suggests we'll probably have to increase it in the future to accommodate some other index a bit larger than *i*. To avoid the cost of the memory allocation for the second (anticipated) expansion, we'll increase the size of the `DynArray` now by *more* than is required to make *i* valid, and we'll hope that future expansions occur within the range we have thereby provided for. For example, we could write `DynArray`::`operator[]` like this:

Click here to view code image

```
template<class T>
T& DynArray<T>::operator[](int index)
{
  if (index < 0) throw an exception;

  if (index > the current maximum index value) {
    int diff = index - the current maximum index value;

    call new to allocate enough additional memory so that
    index+diff is valid;
  }

  return the indexth element of the array;
}
```

This function allocates twice as much memory as needed each time the array must be extended. If we look again at the usage scenario we saw earlier, we note that the `DynArray` must allocate additional memory only once, even though its logical size is extended twice:

Click here to view code image

```
DynArray<double> a;              // only a[0] is valid

a[22] = 3.5;                     // new is called to expand
                                 // a's storage through
                                 // index 44; a's logical
                                 // size becomes 23

a[32] = 0;                       // a's logical size is
                                 // changed to allow a[32],
                                 // but new isn't called
```

If `a` needs to be extended again, that extension, too, will be inexpensive, provided the new maximum index is no greater than 44.

There is a common theme running through this Item, and that's that greater speed can often be purchased at a cost of increased memory usage. Keeping track of running minima, maxima, and averages requires extra space, but it saves time. Caching results necessitates greater memory usage but reduces the time needed to regenerate the results once they've been cached. Prefetching demands a place to put the things that are prefetched, but it reduces the time needed to access those things. The story is as old as Computer Science: you can often trade space for time. (Not always, however. Using larger objects means fewer fit on a virtual memory or cache page. In rare cases, making objects bigger *reduces* the performance of your software, because your paging activity increases, your cache hit rate decreases, or both. How do you find out if you're suffering from such problems? You profile, profile, profile (see Item 16).)

The advice I proffer in this Item — that you amortize the cost of anticipated computations through over-eager strategies like caching and prefetching — is not contradictory to the advice on lazy evaluation I put forth in Item 17. Lazy evaluation is a technique for improving the efficiency of programs when you must support operations whose results are *not always* needed. Over-eager evaluation is a technique for improving the efficiency of programs when you must support operations whose results are *almost always* needed or whose results are often needed more than once. Both are more difficult to implement than run-of-the-mill eager evaluation, but both can yield significant performance improvements in programs whose behavioral characteristics justify the extra programming effort.

Item 19: Understand the origin of temporary objects

When programmers speak amongst themselves, they often refer to variables that are needed for only a short while as "temporaries." For example, in this `swap` routine,

```
template<class T>
void swap(T& object1, T& object2)
{
  T temp = object1;
  object1 = object2;
  object2 = temp;
}
```

it's common to call `temp` a "temporary." As far as C++ is concerned, however, `temp` is not a temporary at all. It's simply an object local to a function.

True temporary objects in C++ are invisible — they don't appear in your source code. They arise whenever a non-heap object is created but not named. Such *unnamed* objects usually arise in one of two situations: when implicit type conversions are applied to make function calls succeed and when functions return objects. It's important to understand how and why these temporary objects are created and destroyed, because the attendant costs of their construction and destruction can have a noticeable impact on the performance of your programs.

Consider first the case in which temporary objects are created to make function calls succeed. This happens when the type of object passed to a function is not the same as the type of the parameter to which it is being bound. For example, consider a function that counts the number of occurrences of a character in a string:

Click here to view code image

```
// returns the number of occurrences of ch in str
size_t countChar(const string& str, char ch);

char buffer[MAX_STRING_LEN];
char c;

// read in a char and a string; use setw to avoid
// overflowing buffer when reading the string
cin >> c >> setw(MAX_STRING_LEN) >> buffer;

cout << "There are " << countChar(buffer, c)
     << " occurrences of the character " << c
     << " in " << buffer << endl;
```

Look at the call to `countChar`. The first argument passed is a `char` array, but the corresponding function parameter is of type `const string&`. This call can succeed only if the type mismatch can be eliminated, and your compilers will be happy to eliminate it by creating a temporary object of type `string`. That temporary object is initialized by calling the `string` constructor with `buffer` as its argument. The `str` parameter of `countChar` is then bound to this temporary `string` object. When the statement containing the call to `countChar` finishes executing, the temporary object is automatically destroyed.

Conversions such as these are convenient (though dangerous — see Item 5), but from an efficiency point of view, the construction and destruction of a temporary `string` object is an unnecessary expense. There are two general ways to eliminate it. One is to redesign your code so conversions like these can't take place. That strategy is examined in Item 5. An alternative tack is to modify your software so that the conversions are unnecessary. Item 21 describes how you can do that.

These conversions occur only when passing objects by value or when passing to a reference-to-`const` parameter. They do not occur when passing an object to a reference-to-non-`const` parameter. Consider this function:

Click here to view code image

```
void uppercasify(string& str);      // changes all chars in
                                    // str to upper case
```

In the character-counting example, a `char` array could be successfully passed to `countChar`, but here, trying to call `uppercasify` with a `char` array fails:

Click here to view code image

```
char subtleBookPlug[] = "Effective C++";

uppercasify(subtleBookPlug);        // error!
```

No temporary is created to make the call succeed. Why not?

Suppose a temporary were created. Then the temporary would be passed to `uppercasify`, which would modify the temporary so its characters were in upper case. But the actual argument to the function call — `subtleBookPlug` — would *not be affected*; only the temporary `string` object generated from `subtleBookPlug` would be changed. Surely this is not what the programmer intended. That programmer passed `subtleBookPlug` to `uppercasify`, and that programmer expected `subtleBookPlug` to be modified. Implicit type conversion for references-to-non-`const` objects, then, would allow temporary objects to be changed when programmers expected non-temporary objects to be modified. That's why the language prohibits the generation of temporaries for non-`const` reference parameters. Reference-to-`const` parameters don't suffer from this problem, because such parameters, by virtue of being `const`, can't be changed.

The second set of circumstances under which temporary objects are created is when a function returns an object. For instance, `operator+` must return an object that represents the sum of its operands. Given a type `Number`, for example, `operator+` for that type would be declared like this:

Click here to view code image

```
const Number operator+(const Number& lhs,
                       const Number& rhs);
```

The return value of this function is a temporary, because it has no name: it's just the function's return value. You must pay to construct and destruct this object each time you call `operator+`. (For an explanation of why the return value is `const`, see Item 6.)

As usual, you don't want to incur this cost. For this particular function, you can avoid paying by switching to a similar function, `operator+=`; Item 22 tells you about this transformation. For most functions that return objects, however, switching to a different function is not an option and there is no way to avoid the construction and destruction of the return value. At least, there's no way to avoid it *conceptually*. Between concept and reality, however, lies a murky zone called *optimization*, and sometimes you can write your object-returning functions in a way that allows your compilers to optimize temporary objects out of existence. Of these optimizations, the most common and useful is the *return value optimization*, which is the subject of Item 20.

The bottom line is that temporary objects can be costly, so you want to eliminate them whenever you can. More important than this, however, is to train yourself to look for places where temporary objects may be created. Anytime you see a reference-to-`const` parameter, the possibility exists that a temporary will be created to bind to that parameter. Anytime you see a function returning an object, a temporary will be created (and later destroyed). Learn to look for such constructs, and your insight into the cost of "behind the scenes" compiler actions will markedly improve.

Item 20: Facilitate the return value optimization

A function that returns an object is frustrating to efficiency aficionados, because the by-value return, including the constructor and destructor calls it implies (see Item 19), cannot be eliminated. The problem is simple: a function either has to return an object in order to offer correct behavior or it doesn't. If it does, there's no way to get rid of the object being returned. Period.

Consider the `operator*` function for rational numbers:

Click here to view code image

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  ...
  int numerator() const;
  int denominator() const;
};

// For an explanation of why the return value is const,
// see Item 6
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
```

Without even looking at the code for `operator*`, we know it must return an object, because it returns the product of two arbitrary numbers. These are *arbitrary* numbers. How can `operator*` possibly avoid creating a new object to hold their product? It can't, so it must create a new object and return it. C++ programmers have nevertheless expended Herculean efforts in a search for the legendary elimination of the by-value return.

Sometimes people return pointers, which leads to this syntactic travesty:

Click here to view code image

```
// an unreasonable way to avoid returning an object
const Rational * operator*(const Rational& lhs,
                           const Rational& rhs);
Rational a = 10;
Rational b(1, 2);

Rational c = *(a * b);          // Does this look "natural"
                                // to you?
```

It also raises a question. Should the caller delete the pointer returned by the function? The answer is usually yes, and that usually leads to resource leaks.

Other developers return references. That yields an acceptable syntax,

Click here to view code image

```
// a dangerous (and incorrect) way to avoid returning
// an object
const Rational& operator*(const Rational& lhs,
                          const Rational& rhs);
Rational a = 10;
Rational b(1, 2);

Rational c = a * b;            // looks perfectly
reasonable
```

but such functions can't be implemented in a way that behaves correctly. A common attempt looks like this:

Click here to view code image

```
// another dangerous (and incorrect) way to avoid
// returning an object
const Rational& operator*(const Rational& lhs,
                          const Rational& rhs)
{
 Rational result(lhs.numerator() * rhs.numerator(),
                 lhs.denominator() * rhs.denominator());
 return result;
}
```

This function returns a reference to an object that no longer exists. In particular, it returns a reference to the local object `result`, but `result` is automatically destroyed when `operator*` is exited. Returning a reference to an object that's been destroyed is hardly useful.

Trust me on this: some functions (`operator*` among them) just have to return objects. That's the way it is. Don't fight it. You can't win.

That is, you can't win in your effort to eliminate by-value returns from functions that require them. But that's the wrong war to wage. From an efficiency point of view, you shouldn't care that a function returns an object, you should only care about the *cost* of that object. What you need to do is channel your efforts into finding a way to reduce the *cost* of returned objects, not to eliminate the objects themselves (which we now recognize is a futile quest). If no cost is associated with such objects, who cares how many get created?

It is frequently possible to write functions that return objects in such a way that compilers can eliminate the cost of the temporaries. The trick is to return *constructor arguments* instead of objects, and you can do it like this:

Click here to view code image

```
// an efficient and correct way to implement a
// function that returns an object
const Rational operator*(const Rational& lhs,
                         const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}
```

Look closely at the expression being returned. It looks like you're calling a `Rational` constructor, and in fact you are. You're creating a temporary `Rational` object through this expression,

Click here to view code image

```
Rational(lhs.numerator() * rhs.numerator(),
         lhs.denominator() * rhs.denominator());
```

and it is this temporary object the function is copying for its return value.

This business of returning constructor arguments instead of local objects doesn't appear to have bought you a lot, because you still have to pay for the construction and destruction of the temporary created inside the function, and you still have to pay for the construction and destruction of the object the function returns. But you have gained something. The rules for C++ allow compilers to optimize temporary objects out of existence. As a result, if you call `operator*` in a context like this,

Click here to view code image

```
Rational a = 10;
Rational b(1, 2);

Rational c = a * b;            // operator* is called here
```

your compilers are allowed to eliminate both the temporary inside `operator*` *and* the temporary returned by `operator*`. They can construct the object defined by the `return` expression *inside the memory allotted for the object* `c`. If your compilers do this, the total cost of temporary objects as a result of your calling `operator*` is zero: no temporaries are created. Instead, you pay for only one constructor call — the one to create `c`. Furthermore, you can't do any better than this, because `c` is a named object, and named objects can't be eliminated (see also Item 22).[†] You can, however, eliminate the overhead of the call to `operator*` by declaring that function `inline`:

Click here to view code image

```
// the most efficient way to write a function returning
// an object
inline const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}
```

"Yeah, yeah," you mutter, "optimization, schmoptimization. Who cares what compilers *can* do? I want to know what they *do* do. Does any of this nonsense work with real compilers?" It does. This particular optimization — eliminating a local temporary by using a function's return location (and possibly replacing that with an object at the function's call site) — is both well-known and commonly implemented. It even has a name: the *return value optimization*. In fact, the existence of a name for this optimization may explain why it's so widely available. Programmers looking for a C++ compiler can ask vendors whether the return value optimization is implemented. If one vendor says yes and another says "The what?," the first vendor has a notable competitive advantage. Ah, capitalism. Sometimes you just gotta love it.

Item 21: Overload to avoid implicit type conversions

Here's some code that looks nothing if not eminently reasonable:

Click here to view code image

```
class UPInt {                         // class for unlimited
public:                               // precision integers
  UPInt();
  UPInt(int value);
  ...

};
```

```
// For an explanation of why the return value is const,
// see Item 6
const UPInt operator+(const UPInt& lhs, const UPInt& rhs);

UPInt upi1, upi2;

...

UPInt upi3 = upi1 + upi2;
```

There are no surprises here. `upi1` and `upi2` are both `UPInt` objects, so adding them together just calls `operator+` for `UPInts`.

Now consider these statements:

```
upi3 = upi1 + 10;

upi3 = 10 + upi2;
```

These statements also succeed. They do so through the creation of temporary objects to convert the integer 10 into `UPInts` (see Item 19).

It is convenient to have compilers perform these kinds of conversions, but the temporary objects created to make the conversions work are a cost we may not wish to bear. Just as most people want government benefits without having to pay for them, most C++ programmers want implicit type conversions without incurring any cost for temporaries. But without the computational equivalent of deficit spending, how can we do it?

We can take a step back and recognize that our goal isn't really type conversion, it's being able to make calls to `operator+` with a combination of `UPInt` and `int` arguments. Implicit type conversion happens to be a means to that end, but let us not confuse means and ends. There is another way to make mixed-type calls to `operator+` succeed, and that's to eliminate the need for type conversions in the first place. If we want to be able to add `UPInt` and `int` objects, all we have to do is say so. We do it by declaring *several* functions, each with a different set of parameter types:

Click here to view code image

```
const UPInt operator+(const UPInt& lhs,      // add UPInt
                      const UPInt& rhs);     // and UPInt

const UPInt operator+(const UPInt& lhs,      // add UPInt
                      int rhs);              // and int
```

```
const UPInt operator+(int lhs,               // add int and
                      const UPInt& rhs);     // UPInt

UPInt upi1, upi2;

...

UPInt upi3 = upi1 + upi2;      // fine, no temporary for
                               // upi1 or upi2

upi3 = upi1 + 10;              // fine, no temporary for
                               // upi1 or 10

upi3 = 10 + upi2;              // fine, no temporary for
                               // 10 or upi2
```

Once you start overloading to eliminate type conversions, you run the risk of getting swept up in the passion of the moment and declaring functions like this:

```
const UPInt operator+(int lhs, int rhs);      // error!
```

The thinking here is reasonable enough. For the types `UPInt` and `int`, we want to overload on all possible combinations for `operator+`. Given the three overloadings above, the only one missing is `operator+` taking two `int` arguments, so we want to add it.

Reasonable or not, there are rules to this C++ game, and one of them is that every overloaded operator must take at least one argument of a user-defined type. `int` isn't a user-defined type, so we can't overload an operator taking only arguments of that type. (If this rule didn't exist, programmers would be able to change the meaning of predefined operations, and that would surely lead to chaos. For example, the attempted overloading of `operator+` above would change the meaning of addition on `int`s. Is that really something we want people to be able to do?)

Overloading to avoid temporaries isn't limited to operator functions. For example, in most programs, you'll want to allow a `string` object everywhere a `char*` is acceptable, and vice versa. Similarly, if you're using a numerical class like `complex` (see Item 35), you'll want types like `int` and `double` to be valid anywhere a numerical object is. As a result, any function taking arguments of type `string`, `char*`, `complex`, etc., is a reasonable candidate for overloading to eliminate type conversions.

Still, it's important to keep the 80-20 rule (see Item 16) in mind. There is no point in implementing a slew of overloaded functions unless you have good reason to believe that it will make a noticeable improvement in the overall efficiency of the programs that use them.

Item 22: Consider using *op*= instead of stand-alone *op*

Most programmers expect that if they can say things like these,

```
x = x + y;                     x = x - y;
```

they can also say things like these:

```
x += y;                        x -= y;
```

If `x` and `y` are of a user-defined type, there is no guarantee that this is so. As far as C++ is concerned, there is no relationship between `operator+`, `operator=`, and `operator+=`, so if you want all three operators to exist and to have the expected relationship, you must implement that yourself. Ditto for the operators `-`, `*`, `/`, etc.

A good way to ensure that the natural relationship between the assignment version of an operator (e.g., `operator+=`) and the stand-alone version (e.g., `operator+`) exists is to implement the latter in terms of the former (see also Item 6). This is easy to do:

Click here to view code image

```
class Rational {
public:
  ...
  Rational& operator+=(const Rational& rhs);
  Rational& operator-=(const Rational& rhs);
};

// operator+ implemented in terms of operator+=; see
// Item 6 for an explanation of why the return value is
// const and page 109B for a warning about implementation
const Rational operator+(const Rational& lhs,
                         const Rational& rhs)
{
  return Rational(lhs) += rhs;
}

// operator- implemented in terms of operator -=
const Rational operator-(const Rational& lhs,
                         const Rational& rhs)
{
  return Rational(lhs) -= rhs;
}
```

In this example, operators `+=` and `-=` are implemented (elsewhere) from scratch, and `operator+` and `operator-` call them to provide their own functionality. With this design, only the assignment versions of these operators need to be maintained. Furthermore, assuming the assignment versions of the operators are in the class's public interface, there is never a need for the stand-alone operators to be friends of the class.

If you don't mind putting all stand-alone operators at global scope, you can use templates to eliminate the need to write the stand-alone functions:

Click here to view code image

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{
  return T(lhs) += rhs;         // see discussion below
}
template<class T>
const T operator-(const T& lhs, const T& rhs)
{
  return T(lhs) -= rhs;         // see discussion below
}
...
```

With these templates, as long as an assignment version of an operator is defined for some type T, the corresponding stand-alone operator will automatically be generated if it's needed.

All this is well and good, but so far we have failed to consider the issue of efficiency, and efficiency is, after all, the topic of this chapter. Three aspects of efficiency are worth noting here. The first is that, in general, assignment versions of operators are more efficient than stand-alone versions, because stand-alone versions must typically return a new object, and that costs us the construction and destruction of a temporary (see Items 19 and 20). Assignment versions of operators write to their left-hand argument, so there is no need to generate a temporary to hold the operator's return value.

The second point is that by offering assignment versions of operators as well as stand-alone versions, you allow *clients* of your classes to make the difficult trade-off between efficiency and convenience. That is, your clients can decide whether to write their code like this,

Click here to view code image

```
Rational a, b, c, d, result;
...
result = a + b + c + d;         // probably uses 3 temporary
                                // objects, one for each call
                                // to operator+
```

or like this:

Click here to view code image

```
result = a;                     // no temporary needed
result += b;                    // no temporary needed
result += c;                    // no temporary needed
result += d;                    // no temporary needed
```

The former is easier to write, debug, and maintain, and it offers acceptable performance about 80% of the time (see Item 16). The latter is more efficient, and, one supposes, more intuitive for assembly language programmers. By offering both options, you let clients develop and debug code using the easier-to-read stand-alone operators while still reserving the right to replace them with the more efficient assignment versions of the operators. Furthermore, by implementing the stand-alones in terms of the assignment versions, you ensure that when clients switch from one to the other, the semantics of the operations remain constant.

The final efficiency observation concerns implementing the standalone operators. Look again at the implementation for `operator+`:

Click here to view code image

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{ return T(lhs) += rhs; }
```

The expression `T(lhs)` is a call to `T`'s copy constructor. It creates a temporary object whose value is the same as that of `lhs`. This temporary is then used to invoke `operator+=` with `rhs`, and the result of that operation is returned from `operator+`.[†] This code seems unnecessarily cryptic. Wouldn't it be better to write it like this?

Click here to view code image

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{
 T result(lhs);                 // copy lhs into result
 return result += rhs;          // add rhs to it and return
}
```

This template is almost equivalent to the one above, but there is a crucial difference. This second template contains a named object, `result`. The fact that this object is named means that the return value optimization (see Item 20) was, until relatively recently, unavailable for this implementation of `operator+` (see the footnote on page 104B). The first implementation has *always* been eligible for the return value optimization, so the odds may be better that the compilers you use will generate optimized code for it.

Now, truth in advertising compels me to point out that the expression

```
return T(lhs) += rhs;
```

is more complex than most compilers are willing to subject to the return value optimization. The first implementation above may thus cost you one temporary object within the function, just as you'd pay for using the named object `result`. However, the fact remains that unnamed objects have historically been easier to eliminate than named objects, so when faced with a choice between a named object and a temporary object, you may be better off using the temporary. It should never cost you more than its named colleague, and, especially with older compilers, it may cost you less.

All this talk of named objects, unnamed objects, and compiler optimizations is interesting, but let us not forget the big picture. The big picture is that assignment versions of operators (such as `operator+=`) tend to be more efficient than stand-alone versions of those operators (e.g. `operator+`). As a library designer, you should offer both, and as an application developer, you should consider using assignment versions of operators instead of stand-alone versions whenever performance is at a premium.

Item 23: Consider alternative libraries

Library design is an exercise in compromise. The ideal library is small, fast, powerful, flexible, extensible, intuitive, universally available, well supported, free of use restrictions, and bug-free. It is also nonexistent. Libraries optimized for size and speed are typically not portable. Libraries with rich functionality are rarely intuitive. Bug-free libraries are limited in scope. In the real world, you can't have everything; something always has to give.

Different designers assign different priorities to these criteria. They thus sacrifice different things in their designs. As a result, it is not uncommon for two libraries offering similar functionality to have quite different performance profiles.

As an example, consider the iostream and stdio libraries, both of which should be available to every C++ programmer. The iostream library has several advantages over its C counterpart. It's type-safe, for example, and it's extensible. In terms of efficiency, however, the iostream library generally suffers in comparison with stdio, because stdio usually results in executables that are both smaller and faster than those arising from iostreams.

Consider first the speed issue. One way to get a feel for the difference in performance between iostreams and stdio is to run benchmark applications using both libraries. Now, it's important to bear in mind that benchmarks lie. Not only is it difficult to come up with a set of inputs that correspond to "typical" usage of a program or library, it's also useless unless you have a reliable way of determining how "typical" you or your clients are. Nevertheless, benchmarks can provide *some* insight into the comparative performance of different approaches to a problem, so though it would be foolish to rely on them completely, it would also be foolish to ignore them.

Let's examine a simple-minded benchmark program that exercises only the most rudimentary I/O functionality. This program reads 30,000 floating point numbers from standard input and writes them to standard output in a fixed format. The choice between the iostream and stdio libraries is made during compilation and is determined by the preprocessor symbol `STDIO`. If this symbol is defined, the stdio library is used, otherwise the iostream library is employed.

Click here to view code image

```
#ifdef STDIO
#include <stdio.h>
```

```
#else
#include <iostream>
#include <iomanip>
using namespace std;
#endif


const int VALUES = 30000;      // # of values to read/write

int main()
{
  double d;

  for (int n = 1; n <= VALUES; ++n) {
#ifdef STDIO
   scanf("%lf", &d);
   printf("%10.5f", d);
#else
   cin >> d;
   cout << setw(10)                    // set field width
        << setprecision(5)             // set decimal
places
        << setiosflags(ios::showpoint) // keep trailing 0s
        << setiosflags(ios::fixed)     // use these
settings
        << d;
#endif

   if (n % 5 == 0) {
#ifdef STDIO
     printf("\n");
#else
     cout << '\n';
#endif
   }
 }

 return 0;
}
```

When this program is given the natural logarithms of the positive integers as input, it produces output like this:

Click here to view code image

```
0.00000   0.69315   1.09861   1.38629    1.60944
```

```
1.79176   1.94591   2.07944   2.19722    2.30259
2.39790   2.48491   2.56495   2.63906    2.70805
2.77259   2.83321   2.89037   2.94444    2.99573
3.04452   3.09104   3.13549   3.17805    3.21888
```

Such output demonstrates, if nothing else, that it's possible to produce fixed-format I/O using iostreams. Of course,

Click here to view code image

```
cout << setw(10)
     << setprecision(5)
     << setiosflags(ios::showpoint)
     << setiosflags(ios::fixed)
     << d;
```

is nowhere near as easy to type as

```
printf("%10.5f", d);
```

but `operator<<` is both type-safe and extensible, and `printf` is neither.

I have run this program on several combinations of machines, operating systems, and compilers, and in every case the stdio version has been faster. Sometimes it's been only a little faster (about 20%), sometimes it's been substantially faster (nearly 200%), but I've never come across an iostream implementation that was as fast as the corresponding stdio implementation. In addition, the size of this trivial program's executable using stdio tends to be smaller (sometimes *much* smaller) than the corresponding program using iostreams. (For programs of a realistic size, this difference is rarely significant.)

Bear in mind that any efficiency advantages of stdio are highly implementation-dependent, so future implementations of systems I've tested or existing implementations of systems I haven't tested may show a negligible performance difference between iostreams and stdio. In fact, one can reasonably hope to discover an iostream implementation that's *faster* than stdio, because iostreams determine the types of their operands during compilation, while stdio functions typically parse a format string at runtime.

The contrast in performance between iostreams and stdio is just an example, however, it's not the main point. The main point is that different libraries offering similar functionality often feature different performance trade-offs, so once you've identified the bottlenecks in your software (via profiling —

see Item 16), you should see if it's possible to remove those bottlenecks by replacing one library with another. If your program has an I/O bottleneck, for example, you might consider replacing iostreams with stdio, but if it spends a significant portion of its time on dynamic memory allocation and deallocation, you might see if there are alternative implementations of `operator new` and `operator delete` available (see Item 8). Because different libraries embody different design decisions regarding efficiency, extensibility, portability, type safety, and other issues, you can sometimes significantly improve the efficiency of your software by switching to libraries whose designers gave more weight to performance considerations than to other factors.

Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI

C++ compilers must find a way to implement each feature in the language. Such implementation details are, of course, compiler-dependent, and different compilers implement language features in different ways. For the most part, you need not concern yourself with such matters. However, the implementation of some features can have a noticeable impact on the size of objects and the speed at which member functions execute, so for those features, it's important to have a basic understanding of what compilers are likely to be doing under the hood. The foremost example of such a feature is virtual functions.

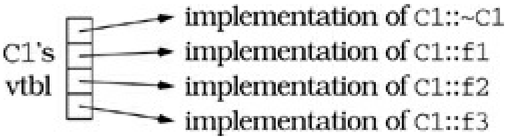
When a virtual function is called, the code executed must correspond to the dynamic type of the object on which the function is invoked; the type of the pointer or reference to the object is immaterial. How can compilers provide this behavior efficiently? Most implementations use *virtual tables* and *virtual table pointers*. Virtual tables and virtual table pointers are commonly referred to as *vtbls* and *vptrs*, respectively.

A vtbl is usually an array of pointers to functions. (Some compilers use a form of linked list instead of an array, but the fundamental strategy is the same.) Each class in a program that declares or inherits virtual functions has its own vtbl, and the entries in a class's vtbl are pointers to the implementations of the virtual functions for that class. For example, given a class definition like this,

Click here to view code image

```
class C1 {
public:
  C1();

  virtual ~C1();
  virtual void f1();
  virtual int f2(char c) const;
  virtual void f3(const string& s);

  void f4() const;

  ...
};
```

`C1`'s virtual table array will look something like this:



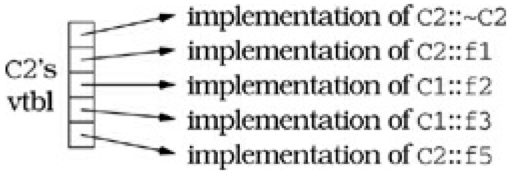
Note that the nonvirtual function `f4` is not in the table, nor is `C1`'s constructor. Nonvirtual functions — including constructors, which are by definition nonvirtual — are implemented just like ordinary C functions, so there are no special performance considerations surrounding their use.

If a class `C2` inherits from `C1`, redefines some of the virtual functions it inherits, and adds some new ones of its own,

Click here to view code image

```
class C2: public C1 {
public:
  C2();                         // nonvirtual function
  virtual ~C2();                // redefined function
  virtual void f1();            // redefined function
  virtual void f5(char *str);   // new virtual function
  ...
};
```

its virtual table entries point to the functions that are appropriate for objects of its type. These entries include pointers to the `C1` virtual functions that `C2` chose not to redefine:



This discussion brings out the first cost of virtual functions: you have to set aside space for a virtual table for each class that contains virtual functions. The size of a class's vtbl is proportional to the number of virtual functions declared for that class (including those it inherits from its base classes). There should be only one virtual table per class, so the total amount of space required for virtual tables is not usually significant, but if you have a large number of classes or a large number of virtual functions in each class, you may find that the vtbls take a significant bite out of your address space.

Because you need only one copy of a class's vtbl in your programs, compilers must address a tricky problem: where to put it. Most programs and libraries are created by linking together many object files, but each object file is generated independently of the others. Which object file should contain the vtbl for any given class? You might think to put it in the object file containing `main`, but libraries have no `main`, and at any rate the source file containing `main` may make no mention of many of the classes requiring vtbls. How could compilers then know which vtbls they were supposed to create?

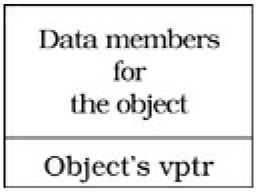
A different strategy must be adopted, and compiler vendors tend to fall into two camps. For vendors who provide an integrated environment containing both compiler and linker, a brute-force strategy is to generate a copy of the vtbl in each object file that might need it. The linker then strips out duplicate copies, leaving only a single instance of each vtbl in the final executable or library.

A more common design is to employ a heuristic to determine which object file should contain the vtbl for a class. Usually this heuristic is as follows: a class's vtbl is generated in the object file containing the definition (i.e., the body) of the first non-inline non-pure virtual function in that class. Thus, the vtbl for class `C1` above would be placed in the object file containing the definition of `C1::~C1` (provided that function wasn't `inline`), and the vtbl for class `C2` would be placed in the object file containing the definition of `C2::~C2` (again, provided that function wasn't `inline`).

In practice, this heuristic works well, but you can get into trouble if you go overboard on declaring virtual functions `inline`. If all virtual functions in a class are declared `inline`, the heuristic fails, and most heuristic-based implementations then generate a copy of the class's vtbl in *every object file* that uses it. In large systems, this can lead to programs containing hundreds or thousands of copies of a class's vtbl! Most compilers following this heuristic give you some way to control vtbl generation manually, but a better solution to this problem is to avoid declaring virtual functions `inline`. As we'll see below, there are good reasons why present compilers typically ignore the `inline` directive for virtual functions, anyway.

Virtual tables are half the implementation machinery for virtual functions, but by themselves they are useless. They become useful only when there is some way of indicating which vtbl corresponds to each object, and it is the job of the virtual table pointer to establish that correspondence.

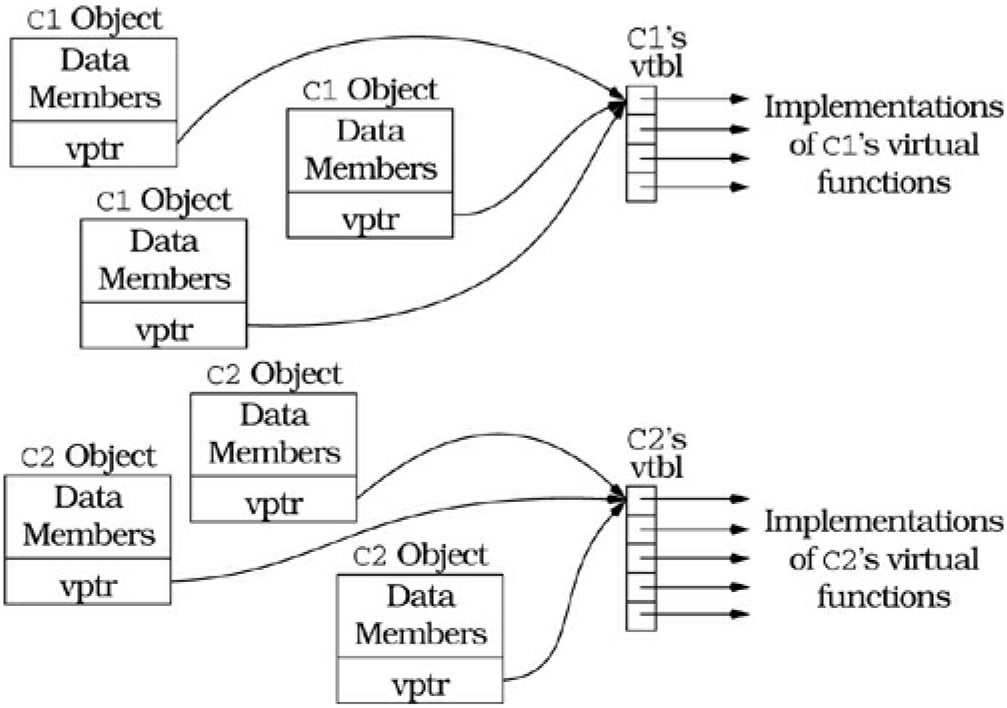
Each object whose class declares virtual functions carries with it a hidden data member that points to the virtual table for that class. This hidden data member — the *vptr* — is added by compilers at a location in the object known only to the compilers. Conceptually, we can think of the layout of an object that has virtual functions as looking like this:



This picture shows the vptr at the end of the object, but don't be fooled: different compilers put them in different places. In the presence of inheritance, an object's vptr is often surrounded by data members. Multiple inheritance complicates this picture, but we'll deal with that a bit later. At this point, simply note the second cost of virtual functions: you have to pay for an extra pointer inside each object that is of a class containing virtual functions.

If your objects are small, this can be a significant cost. If your objects contain, on average, four bytes of member data, for example, the addition of a vptr can *double* their size (assuming four bytes are devoted to the vptr). On systems with limited memory, this means the number of objects you can create is reduced. Even on systems with unconstrained memory, you may find that the performance of your software decreases, because larger objects mean fewer fit on each cache or virtual memory page, and that means your paging activity will probably increase.

Suppose we have a program with several objects of types `C1` and `C2`. Given the relationships among objects, vptrs, and vtbls that we have just seen, we can envision the objects in our program like this:


C1 Object
Data Members
vptr
C1 Object
Data Members
vptr
C1 Object
Data Members
vptr
C1's vtbl
Implementations of C1's virtual functions
C2 Object
Data Members
vptr
C2 Object
Data Members
vptr
C2 Object
Data Members
vptr
C2's vtbl
Implementations of C2's virtual functions

Now consider this program fragment:

```
void makeACall(C1 *pC1)
{
  pC1->f1();
}
```

This is a call to the virtual function `f1` through the pointer `pC1`. By looking only at this code, there is no way to know which `f1` function — `C1::f1` or `C2::f1` — should be invoked, because `pC1` might point to a `C1` object or to a `C2` object. Your compilers must nevertheless generate code for the call to `f1` inside `makeACall`, and they must ensure that the correct function is called, no matter what `pC1` points to. They do this by generating code to do the following:

1. Follow the object's vptr to its vtbl. This is a simple operation, because the compilers know where to look inside the object for the vptr. (After all, they put it there.) As a result, this costs only an offset adjustment (to

get to the vptr) and a pointer indirection (to get to the vtbl).

2. Find the pointer in the vtbl that corresponds to the function being called (`f1` in this example). This, too, is simple, because compilers assign each virtual function a unique index within the table. The cost of this step is just an offset into the vtbl array.
3. Invoke the function pointed to by the pointer located in step 2.

If we imagine that each object has a hidden member called `vptr` and that the vtbl index of function `f1` is `i`, the code generated for the statement

```
pC1->f1();
```

is

Click here to view code image

```
(*pC1->vptr[i])(pC1);  // call the function pointed to by
the
                       // i-th entry in the vtbl pointed to
                       // by pC1->vptr; pC1 is passed to
the
                       // function as the "this" pointer
```

This is almost as efficient as a non-virtual function call: on most machines it executes only a few more instructions. The cost of calling a virtual function is thus basically the same as that of calling a function through a function pointer. Virtual functions *per se* are not usually a performance bottleneck.

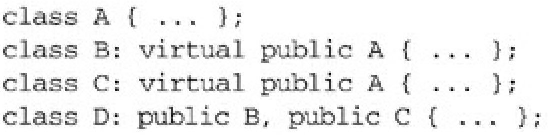
The real runtime cost of virtual functions has to do with their interaction with inlining. For all practical purposes, virtual functions aren't inlined. That's because "inline" means "during compilation, replace the call site with the body of the called function," but "virtual" means "wait until runtime to see which function is called." If your compilers don't know which function will be called at a particular call site, you can understand why they won't inline that function call. This is the third cost of virtual functions: you effectively give up inlining. (Virtual functions *can* be inlined when invoked through *objects*, but most virtual function calls are made through *pointers* or *references* to objects, and such calls are not inlined. Because such calls are the norm, virtual functions are effectively not inlined.)

Everything we've seen so far applies to both single and multiple inheritance, but when multiple inheritance enters the picture, things get more complex.

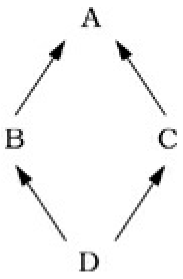
There is no point in dwelling on details, but with multiple inheritance, offset calculations to find vptrs within objects become more complicated; there are multiple vptrs within a single object (one per base class); and special vtbls must be generated for base classes in addition to the stand-alone vtbls we have discussed. As a result, both the per-class and the per-object space overhead for virtual functions increases, and the runtime invocation cost grows slightly, too.

Multiple inheritance often leads to the need for virtual base classes. Without virtual base classes, if a derived class has more than one inheritance path to a base class, the data members of that base class are replicated within each derived class object, one copy for each path between the derived class and the base class. Such replication is almost never what programmers want, and making base classes virtual eliminates the replication. Virtual base classes may incur a cost of their own, however, because implementations of virtual base classes often use pointers to virtual base class parts as the means for avoiding the replication, and one or more of those pointers may be stored inside your objects.

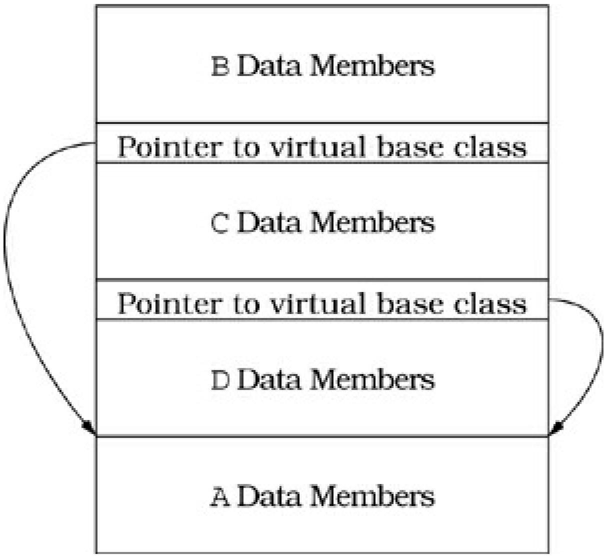
For example, consider this, which I generally call "the dreaded multiple inheritance diamond:"



```
class A { ... };
class B: virtual public A { ... };
class C: virtual public A { ... };
class D: public B, public C { ... };
```

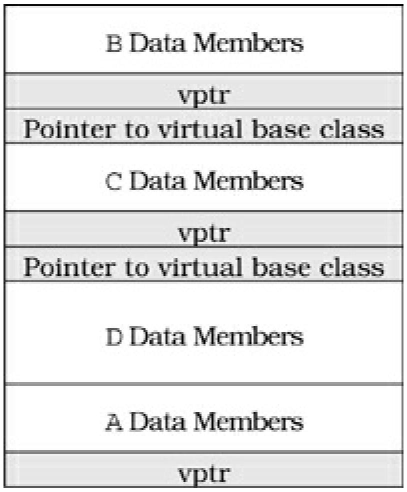


Here `A` is a virtual base class because `B` and `C` virtually inherit from it. With some compilers (especially older compilers), the layout for an object of type `D` is likely to look like this:


B Data Members
Pointer to virtual base class
C Data Members
Pointer to virtual base class
D Data Members
A Data Members

It seems a little strange to place the base class data members at the end of the object, but that's often how it's done. Of course, implementations are free to organize memory any way they like, so you should never rely on this picture for anything more than a conceptual overview of how virtual base classes may lead to the addition of hidden pointers to your objects. Some implementations add fewer pointers, and some find ways to add none at all. (Such implementations make the vptr and vtbl serve double duty).

If we combine this picture with the earlier one showing how virtual table pointers are added to objects, we realize that if the base class `A` in the hierarchy on page 119 has any virtual functions, the memory layout for an object of type `D` could look like this:


B Data Members
vptr
Pointer to virtual base class
C Data Members
vptr
Pointer to virtual base class
D Data Members
A Data Members
vptr

Here I've shaded the parts of the object that are added by compilers. The picture may be misleading, because the ratio of shaded to unshaded areas is determined by the amount of data in your classes. For small classes, the relative overhead is large. For classes with more data, the relative overhead is less significant, though it is typically noticeable.

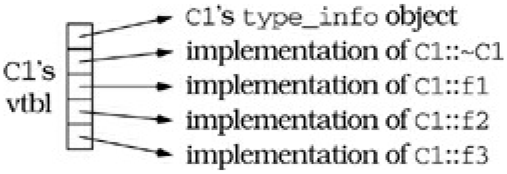
An oddity in the above diagram is that there are only three vptrs even though four classes are involved. Implementations are free to generate four vptrs if they like, but three suffice (it turns out that `B` and `D` can share a vptr), and most implementations take advantage of this opportunity to reduce the compiler-generated overhead.

We've now seen how virtual functions make objects larger and preclude inlining, and we've examined how multiple inheritance and virtual base classes can also increase the size of objects. Let us therefore turn to our final topic, the cost of runtime type identification (RTTI).

RTTI lets us discover information about objects and classes at runtime, so there has to be a place to store the information we're allowed to query. That information is stored in an object of type `type_info`, and you can access the `type_info` object for a class by using the `typeid` operator.

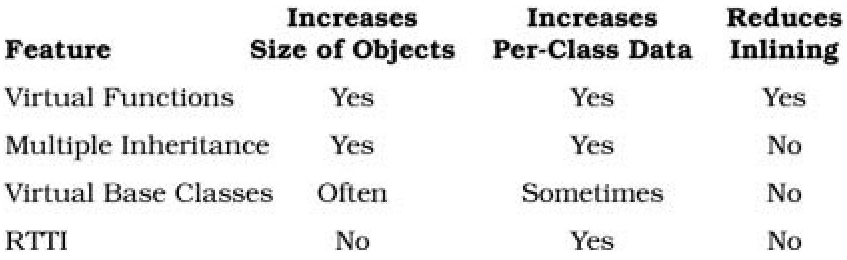
There only needs to be a single copy of the RTTI information for each class, but there must be a way to get to that information for any object. Actually, that's not quite true. The language specification states that we're guaranteed accurate information on an object's dynamic type only if that type has at least one virtual function. This makes RTTI data sound a lot like a virtual function table. We need only one copy of the information per class, and we need a way to get to the appropriate information from any object containing a virtual function. This parallel between RTTI and virtual function tables is no accident: RTTI was designed to be implementable in terms of a class's vtbl.

For example, index 0 of a vtbl array might contain a pointer to the `type_info` object for the class corresponding to that vtbl. The vtbl for class `C1` on page 114 would then look like this:



With this implementation, the space cost of RTTI is an additional entry in each class vtbl plus the cost of the storage for the `type_info` object for each class. Just as the memory for virtual tables is unlikely to be noticeable for most applications, however, you're unlikely to run into problems due to the size of `type_info` objects.

The following table summarizes the primary costs of virtual functions, multiple inheritance, virtual base classes, and RTTI:



Feature	Increases Size of Objects	Increases Per-Class Data	Reduces Inlining
Virtual Functions	Yes	Yes	Yes
Multiple Inheritance	Yes	Yes	No
Virtual Base Classes	Often	Sometimes	No
RTTI	No	Yes	No

Some people look at this table and are aghast. "I'm sticking with C!", they declare. Fair enough. But remember that each of these features offers functionality you'd otherwise have to code by hand. In most cases, your manual approximation would probably be less efficient and less robust than the compiler-generated code. Using nested `switch` statements or cascading `if-then-else`s to emulate virtual function calls, for example, yields more code than virtual function calls do, and the code runs more slowly, too. Furthermore, you must manually track object types yourself, which means your objects carry around type tags of their own; you thus often fail to gain even the benefit of smaller objects.

It is important to understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI, but it is equally important to understand that if you need the functionality these features offer, you *will* pay for it, one way or another. Sometimes you have legitimate reasons for bypassing the compiler-generated services. For example, hidden vptrs and pointers to virtual base classes can make it difficult to store C++ objects in databases or to move them across process boundaries, so you may wish to emulate these features in a way that makes it easier to accomplish these other tasks. From the point of view of efficiency, however, you are unlikely to do better than the compiler-generated implementations by coding these features yourself.

Techniques

Most of this book is concerned with programming guidelines. Such guidelines are important, but no programmer lives by guidelines alone. According to the old TV show *Felix the Cat*, "Whenever he gets in a fix, he reaches into his bag of tricks." Well, if a cartoon character can have a bag of tricks, so too can C++ programmers. Think of this chapter as a starter set for your bag of tricks.

Some problems crop up repeatedly when designing C++ software. How can you make constructors and non-member functions act like virtual functions? How can you limit the number of instances of a class? How can you prevent objects from being created on the heap? How can you guarantee that they will be created there? How can you create objects that automatically perform some actions anytime some other class's member functions are called? How can you have different objects share data structures while giving clients the illusion that each has its own copy? How can you distinguish between read and write usage of `operator[]`? How can you create a virtual function whose behavior depends on the dynamic types of more than one object?

All these questions (and more) are answered in this chapter, in which I describe proven solutions to problems commonly encountered by C++ programmers. I call such solutions *techniques*, but they're also known as *idioms* and, when documented in a stylized fashion, *patterns*. Regardless of what you call them, the information that follows will serve you well as you engage in the day-to-day skirmishes of practical software development. It should also convince you that no matter what you want to do, there is almost certainly a way to do it in C++.

Item 25: Virtualizing constructors and non-member functions

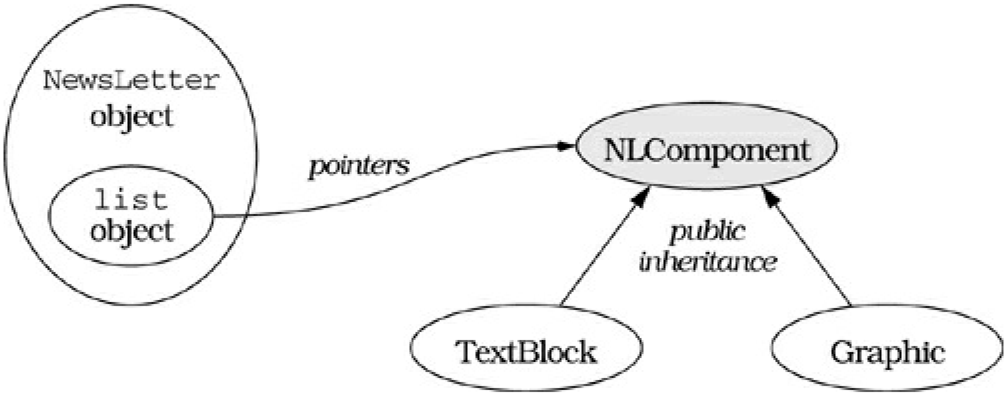
On the face of it, it doesn't make much sense to talk about "virtual constructors." You call a virtual function to achieve type-specific behavior when you have a pointer or reference to an object but you don't know what the real type of the object is. You call a constructor only when you don't yet have an object but you know exactly what type you'd like to have. How, then, can one talk of *virtual* constructors?

It's easy. Though virtual constructors may seem nonsensical, they are remarkably useful. (If you think nonsensical ideas are never useful, how do you explain the success of modern physics?) For example, suppose you write applications for working with newsletters, where a newsletter consists of components that are either textual or graphical. You might organize things this way:

Click here to view code image

```
class NLComponent {              // abstract base class for
public:                          // newsletter components

  ...                            // contains at least one
};                               // pure virtual function

class TextBlock: public NLComponent {
public:
  ...                            // contains no pure virtual
};                               // functions

class Graphic: public NLComponent {
public:
  ...                            // contains no pure virtual
};                               // functions

class NewsLetter {               // a newsletter object
public:                          // consists of a list of
  ...                            // NLComponent objects

private:
  list<NLComponent*> components;
};
```

The classes relate in this way:



The `list` class used inside `NewsLetter` is part of the Standard Template Library, which is part of the standard C++ library (see Item 35). Objects of type `list` behave like doubly linked lists, though they need not be implemented in that way.

`NewsLetter` objects, when not being worked on, would likely be stored on disk. To support the creation of a `Newsletter` from its on-disk representation, it would be convenient to give `NewsLetter` a constructor that takes an `istream`. The constructor would read information from the stream as it created the necessary in-core data structures:

```
class NewsLetter {
public:
  NewsLetter(istream& str);
  ...
};
```

Pseudocode for this constructor might look like this,

Click here to view code image

```
NewsLetter::NewsLetter(istream& str)
{
  while (str) {
    read the next component object from str;

    add the object to the list of this
    newsletter's components;
  }
}
```

or, after moving the tricky stuff into a separate function called `readComponent`, like this:

[Click here to view code image](#)

```
class NewsLetter {
public:
  ...

private:
  // read the data for the next NLComponent from str,
  // create the component and return a pointer to it
  static NLComponent * readComponent(istream& str);
  ...
};

NewsLetter::NewsLetter(istream& str)
{
  while (str) {
    // add the pointer returned by readComponent to the
    // end of the components list; "push_back" is a list
    // member function that inserts at the end of the list
    components.push_back(readComponent(str));
  }
}
```

Consider what `readComponent` does. It creates a new object, either a `TextBlock` or a `Graphic`, depending on the data it reads. Because it creates new objects, it acts much like a constructor, but because it can create different types of objects, we call it a *virtual constructor*. A virtual constructor is a function that creates different types of objects depending on the input it is given. Virtual constructors are useful in many contexts, only one of which is reading object information from disk (or off a network connection or from a tape, etc.).

A particular kind of virtual constructor — the *virtual copy constructor* — is also widely useful. A virtual copy constructor returns a pointer to a new copy of the object invoking the function. Because of this behavior, virtual copy constructors are typically given names like `copySelf`, `cloneSelf`, or, as shown below, just plain `clone`. Few functions are implemented in a more straightforward manner:

[Click here to view code image](#)

```
class NLComponent {
public:
  // declaration of virtual copy constructor
```

```
    virtual NLComponent * clone() const = 0;
    ...

  };

  class TextBlock: public NLComponent {
  public:
    virtual TextBlock * clone() const      // virtual copy
    { return new TextBlock(*this); }       // constructor
    ...

  };

  class Graphic: public NLComponent {
  public:
    virtual Graphic * clone() const        // virtual copy
    { return new Graphic(*this); }         // constructor
    ...

  };
```

As you can see, a class's virtual copy constructor just calls its real copy constructor. The meaning of "copy" is hence the same for both functions. If the real copy constructor performs a shallow copy, so does the virtual copy constructor. If the real copy constructor performs a deep copy, so does the virtual copy constructor. If the real copy constructor does something fancy like reference counting or copy-on-write (see Item 29), so does the virtual copy constructor. Consistency — what a wonderful thing.

Notice that the above implementation takes advantage of a relaxation in the rules for virtual function return types that was adopted relatively recently. No longer must a derived class's redefinition of a base class's virtual function declare the same return type. Instead, if the function's return type is a pointer (or a reference) to a base class, the derived class's function may return a pointer (or reference) to a class derived from that base class. This opens no holes in C++'s type system, and it makes it possible to accurately declare functions such as virtual copy constructors. That's why `TextBlock`'s `clone` can return a `TextBlock*` and `Graphic`'s `clone` can return a `Graphic*`, even though the return type of `NLComponent`'s `clone` is `NLComponent*`.

The existence of a virtual copy constructor in `NLComponent` makes it easy to implement a (normal) copy constructor for `NewsLetter`:

Click here to view code image

```
class NewsLetter {
public:
  NewsLetter(const NewsLetter& rhs);
  ...

private:
  list<NLComponent*> components;
};

NewsLetter::NewsLetter(const NewsLetter& rhs)
{
  // iterate over rhs's list, using each element's
  // virtual copy constructor to copy the element into
  // the components list for this object. For details on
  // how the following code works, see Item 35.
  for (list<NLComponent*>::const_iterator it =
         rhs.components.begin();
       it != rhs.components.end();
       ++it) {

    // "it" points to the current element of
rhs.components,
    // so call that element's clone function to get a copy
    // of the element, and add that copy to the end of
    // this object's list of components
    components.push_back((*it)->clone());
  }
}
```

Unless you are familiar with the Standard Template Library, this code looks bizarre, I know, but the idea is simple: just iterate over the list of components for the `NewsLetter` object being copied, and for each component in the list, call its virtual copy constructor. We need a virtual copy constructor here, because the list contains pointers to `NLComponent` objects, but we know each pointer really points to a `TextBlock` or a `Graphic`. We want to copy whatever the pointer really points to, and the virtual copy constructor does that for us.

Making Non-Member Functions Act Virtual

Just as constructors can't really be virtual, neither can non-member functions. However, just as it makes sense to conceive of functions that construct new objects of different types, it makes sense to conceive of non-member functions whose behavior depends on the dynamic types of their parameters. For example, suppose you'd like to implement output operators for the `TextBlock` and `Graphic` classes. The obvious approach to this problem is to make the output operator virtual. However, the output operator is `operator<<`, and that function takes an `ostream&` as its left-hand argument; that effectively rules out the possibility of making it a member function of the `TextBlock` or `Graphic` classes.

(It can be done, but then look what happens:

Click here to view code image

```
class NLComponent {
public:
  // unconventional declaration of output operator
  virtual ostream& operator<<(ostream& str) const = 0;
  ...
};

class TextBlock: public NLComponent {
public:
  // virtual output operator (also unconventional)
  virtual ostream& operator<<(ostream& str) const;
};

class Graphic: public NLComponent {
public:
  // virtual output operator (still unconventional)
  virtual ostream& operator<<(ostream& str) const;
};

TextBlock t;
Graphic g;

...

t << cout;                        // print t on cout via
                                  // virtual operator<<; note
                                  // unconventional syntax

g << cout;                        // print g on cout via
                                  // virtual operator<<; note
                                  // unconventional syntax
```

Clients must place the stream object on the *right-hand side* of the "`<<`" symbol, and that's contrary to the convention for output operators. To get back to the normal syntax, we must move `operator<<` out of the `TextBlock` and `Graphic` classes, but if we do that, we can no longer declare it virtual.)

An alternate approach is to declare a virtual function for printing (e.g., `print`) and define it for the `TextBlock` and `Graphic` classes. But if we do that, the syntax for printing `TextBlock` and `Graphic` objects is inconsistent with that for the other types in the language, all of which rely on `operator<<` as their output operator.

Neither of these solutions is very satisfying. What we want is a non-member function called `operator<<` that exhibits the behavior of a virtual function like `print`. This description of what we want is in fact very close to a description of how to get it. We define *both* `operator<<` and `print` and have the former call the latter!

Click here to view code image

```
class NLComponent {
public:
  virtual ostream& print(ostream& s) const = 0;
  ...

};

class TextBlock: public NLComponent {
public:
  virtual ostream& print(ostream& s) const;
  ...

};

class Graphic: public NLComponent {
public:
  virtual ostream& print(ostream& s) const;
  ...

};

inline
ostream& operator<<(ostream& s, const NLComponent& c)
{
  return c.print(s);
}
```

Virtual-acting non-member functions, then, are easy. You write virtual functions to do the work, then write a non-virtual function that does nothing but call the virtual function. To avoid incurring the cost of a function call for this syntactic sleight-of-hand, of course, you inline the non-virtual function.

Now that you know how to make non-member functions act virtually on one of their arguments, you may wonder if it's possible to make them act virtually on more than one of their arguments. It is, but it's not easy. How hard is it? Turn to Item 31; it's devoted to that question.

Item 26: Limiting the number of objects of a class

Okay, you're crazy about objects, but sometimes you'd like to bound your insanity. For example, you've got only one printer in your system, so you'd like to somehow limit the number of printer objects to one. Or you've got only 16 file descriptors you can hand out, so you've got to make sure there are never more than that many file descriptor objects in existence. How can you do such things? How can you limit the number of objects?

If this were a proof by mathematical induction, we might start with $n = 1$, then build from there. Fortunately, this is neither a proof nor an induction. Moreover, it turns out to be instructive to begin with $n = 0$, so we'll start there instead. How do you prevent objects from being instantiated at all?

Allowing Zero or One Objects

Each time an object is instantiated, we know one thing for sure: a constructor will be called. That being the case, the easiest way to prevent objects of a particular class from being created is to declare the constructors of that class *private*:

Click here to view code image

```
class CantBeInstantiated {
private:
  CantBeInstantiated();
  CantBeInstantiated(const CantBeInstantiated&);

  ...

};
```

Having thus removed everybody's right to create objects, we can selectively loosen the restriction. If, for example, we want to create a class for printers, but we also want to abide by the constraint that there is only one printer available to us, we can encapsulate the printer object inside a function so that everybody has access to the printer, but only a single printer object is created:

Click here to view code image

```
class PrintJob;                 // forward declaration

class Printer {
public:
  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();

  ...

friend Printer& thePrinter();
private:
  Printer();
  Printer(const Printer& rhs);

  ...

};

Printer& thePrinter()
{
  static Printer p;           // the single printer object
  return p;
}
```

There are three separate components to this design. First, the constructors of the `Printer` class are private. That suppresses object creation. Second, the global function `thePrinter` is declared a friend of the class. That lets `thePrinter` escape the restriction imposed by the private constructors. Finally, `thePrinter` contains a *static* `Printer` object. That means only a single object will be created.

Client code refers to `thePrinter` whenever it wishes to interact with the system's lone printer. By returning a reference to a `Printer` object, `thePrinter` can be used in any context where a `Printer` object itself could be:

[Click here to view code image](#)

```
class PrintJob {
public:
  PrintJob(const string& whatToPrint);
  ...

};

string buffer;

...                               // put stuff in buffer

thePrinter().reset();
thePrinter().submitJob(buffer);
```

It's possible, of course, that `thePrinter` strikes you as a needless addition to the global namespace. "Yes," you may say, "as a global function it looks more like a global variable, but global variables are gauche, and I'd prefer to localize all printer-related functionality inside the `Printer` class." Well, far be it from me to argue with someone who uses words like gauche. `thePrinter` can just as easily be made a static member function of `Printer`, and that puts it right where you want it. It also eliminates the need for a `friend` declaration, which many regard as tacky in its own right. Using a static member function, `Printer` looks like this:

[Click here to view code image](#)

```
class Printer {
public:
  static Printer& thePrinter();
  ...

private:
  Printer();
  Printer(const Printer& rhs);
  ...

};

Printer& Printer::thePrinter()
{
  static Printer p;
  return p;
}
```

Clients must now be a bit wordier when they refer to the printer:

Click here to view code image

```
Printer::thePrinter().reset();
Printer::thePrinter().submitJob(buffer);
```

Another approach is to move `Printer` and `thePrinter` out of the global scope and into a *namespace*. Namespaces are a recent addition to C++. Anything that can be declared at global scope can also be declared in a namespace. This includes classes, structs, functions, variables, objects, typedefs, etc. The fact that something is in a namespace doesn't affect its behavior, but it does prevent name conflicts between entities in different namespaces. By putting the `Printer` class and the `thePrinter` function into a namespace, we don't have to worry about whether anybody else happened to choose the names `Printer` or `thePrinter` for themselves; our namespace prevents name conflicts.

Syntactically, namespaces look much like classes, but there are no public, protected, or private sections; everything is public. This is how we'd put `Printer` and `thePrinter` into a namespace called `PrintingStuff`:

Click here to view code image

```
namespace PrintingStuff {
  class Printer {                 // this class is in the
  public:                         // PrintingStuff
namespace

    void submitJob(const PrintJob& job);
    void reset();
    void performSelfTest();
    ...

  friend Printer& thePrinter();
  private:
    Printer();
    Printer(const Printer& rhs);
    ...

  };

  Printer& thePrinter()           // so is this function
  {
    static Printer p;
    return p;
    }
  }                                      // this is the end of the
                                         // namespace
```

Given this namespace, clients can refer to `thePrinter` using a fullyqualified name (i.e., one that includes the name of the namespace),

[Click here to view code image]

```
PrintingStuff::thePrinter().reset();
PrintingStuff::thePrinter().submitJob(buffer);
```

but they can also employ a *`using` declaration* to save themselves keystrokes:

[Click here to view code image]

```
using PrintingStuff::thePrinter; // import the name
                                 // "thePrinter" from the
                                 // namespace
"PrintingStuff"
                                 // into the current scope

thePrinter().reset();            // now thePrinter can be
thePrinter().submitJob(buffer);  // used as if it were a
                                 // local name
```

There are two subtleties in the implementation of `thePrinter` that are worth exploring. First, it's important that the single `Printer` object be static in a *function* and not in a class. An object that's static in a class is, for all intents and purposes, *always* constructed (and destructed), even if it's never used. In contrast, an object that's static in a function is created the first time through the function, so if the function is never called, the object is never created. (You do, however, pay for a check each time the function is called to see whether the object needs to be created.) One of the philosophical pillars on which C++ was built is the idea that you shouldn't pay for things you don't use, and defining an object like our printer as a static object in a function is one way of adhering to this philosophy. It's a philosophy you should adhere to whenever you can.

There is another drawback to making the printer a class static versus a function static, and that has to do with its time of initialization. We know exactly when a function static is initialized: the first time through the function at the point where the static is defined. The situation with a class static (or, for that matter, a global static, should you be so gauche as to use one) is less well defined. C++ offers certain guarantees regarding the order of initialization of statics within a particular translation unit (i.e., a body of source code that yields a single object file), but it says *nothing* about the initialization order of static objects in different translation units. In practice, this turns out to be a source of countless headaches. Function statics, when they can be made to suffice, allow us to avoid these headaches. In our example here, they can, so why suffer?

The second subtlety has to do with the interaction of inlining and static objects inside functions. Look again at the code for the non-member version of `thePrinter`:

```
Printer& thePrinter()
{
  static Printer p;
  return p;
}
```

Except for the first time through this function (when `p` must be constructed), this is a one-line function — it consists entirely of the statement "`return p;`". If ever there were a good candidate for inlining, this function would certainly seem to be the one. Yet it's not declared `inline`. Why not?

Consider for a moment why you'd declare an object to be static. It's usually because you want only a single copy of that object, right? Now consider what `inline` means. Conceptually, it means compilers should replace each call to the function with a copy of the function body, but for non-member functions, it also means something else. It means the functions in question have *internal linkage*.

You don't ordinarily need to worry about such linguistic mumbo jumbo, but there is one thing you must remember: functions with internal linkage may be duplicated within a program (i.e., the object code for the program may contain more than one copy of each function with internal linkage), and *this duplication includes static objects contained within the functions*. The result? If you create an inline non-member function containing a local static object, you may end up with *more than one copy* of the static object in your program! So don't create inline non-member functions that contain local static data.[†]

But maybe you think this business of creating a function to return a reference to a hidden object is the wrong way to go about limiting the number of objects in the first place. Perhaps you think it's better to simply count the number of objects in existence and throw an exception in a constructor if too many objects are requested. In other words, maybe you think we should handle printer creation like this:

[Click here to view code image]

```
class Printer {
public:
  class TooManyObjects{};        // exception class for use
                                 // when too many objects
                                 // are requested
  Printer();
  ~Printer();

  ...

private:
  static size_t numObjects;

  Printer(const Printer& rhs);   // there is a limit of 1
                                 // printer, so never allow
};                               // copying
```

The idea is to use `numObjects` to keep track of how many `Printer` objects are in existence. This value will be incremented in the class constructor and decremented in its destructor. If an attempt is made to construct too many Printer objects, we throw an exception of type `TooManyObjects`:

[Click here to view code image]

```
// Obligatory definition of the class static
size_t Printer::numObjects = 0;

Printer::Printer()
{
  if (numObjects >= 1) {
    throw TooManyObjects();
  }

  proceed with normal construction here;

  ++numObjects;
}
```

```
Printer::~Printer()
{
  perform normal destruction here;

  --numObjects;
}
```

This approach to limiting object creation is attractive for a couple of reasons. For one thing, it's straightforward — everybody should be able to understand what's going on. For another, it's easy to generalize so that the maximum number of objects is some number other than one.

Contexts for Object Construction

There is also a problem with this strategy. Suppose we have a special kind of printer, say, a color printer. The class for such printers would have much in common with our generic printer class, so of course we'd inherit from it:

```
class ColorPrinter: public Printer {
  ...
};
```

Now suppose we have one generic printer and one color printer in our system:

```
Printer p;
ColorPrinter cp;
```

How many `Printer` objects result from these object definitions? The answer is two: one for `p` and one for the `Printer` part of `cp`. At runtime, a `TooManyObjects` exception will be thrown during the construction of the base class part of `cp`. For many programmers, this is neither what they want nor what they expect. (Designs that avoid having concrete classes inherit from other concrete classes do not suffer from this problem. For details on this design philosophy, see Item 33.)

A similar problem occurs when `Printer` objects are contained inside other objects:

Click here to view code image

```
class CPFMachine {       // for machines that can
private:                 // copy, print, and fax
```

```
  Printer p;            // for printing capabilities
  FaxMachine f;         // for faxing capabilities
  CopyMachine c;        // for copying capabilities

  ...

};

CPFMachine m1;          // fine

CPFMachine m2;          // throws TooManyObjects exception
```

The problem is that `Printer` objects can exist in three different contexts: on their own, as base class parts of more derived objects, and embedded inside larger objects. The presence of these different contexts significantly muddies the waters regarding what it means to keep track of the "number of objects in existence," because what you consider to be the existence of an object may not jibe with your compilers'.

Often you will be interested only in allowing objects to exist on their own, and you will wish to limit the number of *those* kinds of instantiations. That restriction is easy to satisfy if you adopt the strategy exemplified by our original `Printer` class, because the `Printer` constructors are private, and (in the absence of `friend` declarations) classes with private constructors can't be used as base classes, nor can they be embedded inside other objects.

The fact that you can't derive from classes with private constructors leads to a general scheme for preventing derivation, one that doesn't necessarily have to be coupled with limiting object instantiations. Suppose, for example, you have a class, `FSA`, for representing finite state automata. (Such state machines are useful in many contexts, among them user interface design.) Further suppose you'd like to allow any number of `FSA` objects to be created, but you'd also like to ensure that no class ever inherits from `FSA`. (One reason for doing this might be to justify the presence of a nonvirtual destructor in `FSA`. As Item 24 explains, classes without virtual functions yield smaller objects than do equivalent classes with virtual functions.) Here's how you can design `FSA` to satisfy both criteria:

Click here to view code image

```
class FSA {
public:
    // pseudo-constructors
    static FSA * makeFSA();
    static FSA * makeFSA(const FSA& rhs);
    ...

  private:
    FSA();
    FSA(const FSA& rhs);
    ...
  };

  FSA * FSA::makeFSA()
  { return new FSA(); }

  FSA * FSA::makeFSA(const FSA& rhs)
  { return new FSA(rhs); }
```

Unlike the `thePrinter` function that always returned a reference to a single object, each `makeFSA` pseudo-constructor returns a pointer to a unique object. That's what allows an unlimited number of `FSA` objects to be created.

This is nice, but the fact that each pseudo-constructor calls `new` implies that callers will have to remember to call `delete`. Otherwise a resource leak will be introduced. Callers who wish to have `delete` called automatically when the current scope is exited can store the pointer returned from `makeFSA` in an `auto_ptr` object (see Item 9); such objects automatically delete what they point to when they themselves go out of scope:

Click here to view code image

```
  // indirectly call default FSA constructor
  auto_ptr<FSA> pfsa1(FSA::makeFSA());

  // indirectly call FSA copy constructor
  auto_ptr<FSA> pfsa2(FSA::makeFSA(*pfsa1));

  ...              // use pfsa1 and pfsa2 as normal pointers,
                   // but don't worry about deleting them
```

Allowing Objects to Come and Go

We now know how to design a class that allows only a single instantiation, we know that keeping track of the number of objects of a particular class is complicated by the fact that object constructors are called in three different contexts, and we know that we can eliminate the confusion surrounding object counts by making constructors private. It is worthwhile to make one final observation. Our use of the `thePrinter` function to encapsulate access to a single object limits the number of `Printer` objects to one, but it also limits us to a single `Printer` object for each run of the program. As a result, it's not possible to write code like this:

```
create Printer object p1;

use p1;

destroy p1;

create Printer object p2;

use p2;

destroy p2;

...
```

This design never instantiates more than a single `Printer` object at a time, but it does use different `Printer` objects in different parts of the program. It somehow seems unreasonable that this isn't allowed. After all, at no point do we violate the constraint that only one printer may exist. Isn't there a way to make this legal?

There is. All we have to do is combine the object-counting code we used earlier with the pseudo-constructors we just saw:

Click here to view code image

```
class Printer {
public:
  class TooManyObjects{};

  // pseudo-constructor
  static Printer * makePrinter();

  ~Printer();
  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
  ...
private:
  static size_t numObjects;

  Printer();

  Printer(const Printer& rhs);  // we don't define this
};                              // function, because we'll
                                // never allow copying

// Obligatory definition of class static
size_t Printer::numObjects = 0;

Printer::Printer()
{
  if (numObjects >= 1) {
    throw TooManyObjects();
  }

  proceed with normal object construction here;

  ++numObjects;
}

Printer * Printer::makePrinter()
{ return new Printer; }
```

If the notion of throwing an exception when too many objects are requested strikes you as unreasonably harsh, you could have the pseudo-constructor return a null pointer instead. Clients would then have to check for this before doing anything with it, of course.

Clients use this `Printer` class just as they would any other class, except they must call the pseudo-constructor function instead of the real constructor:

Click here to view code image

```
Printer p1;                    // error! default ctor is
                               // private

Printer *p2 =
  Printer::makePrinter();      // fine, indirectly calls
                               // default ctor

Printer p3 = *p2;              // error! copy ctor is
```

```
                              // private

p2->performSelfTest();        // all other functions are
p2->reset();                  // called as usual

...

delete p2;                    // avoid resource leak; this
                              // would be unnecessary if
                              // p2 were an auto_ptr
```

This technique is easily generalized to any number of objects. All we have to do is replace the hard-wired constant 1 with a class-specific value, then lift the restriction against copying objects. For example, the following revised implementation of our `Printer` class allows up to 10 `Printer` objects to exist:

Click here to view code image

```
class Printer {
public:
  class TooManyObjects{};

  // pseudo-constructors
  static Printer * makePrinter();
  static Printer * makePrinter(const Printer& rhs);

  ...

private:
  static size_t numObjects;
  static const size_t maxObjects = 10;   // see below

  Printer();
  Printer(const Printer& rhs);
};

// Obligatory definitions of class statics
size_t Printer::numObjects = 0;
const size_t Printer::maxObjects;

Printer::Printer()
{
  if (numObjects >= maxObjects) {
    throw TooManyObjects();
  }
  ...

}

Printer::Printer(const Printer& rhs)
{
  if (numObjects >= maxObjects) {
    throw TooManyObjects();
  }

  ...

}

Printer * Printer::makePrinter()
{ return new Printer; }

Printer * Printer::makePrinter(const Printer& rhs)
{ return new Printer(rhs); }
```

Don't be surprised if your compilers get all upset about the declaration of `Printer`::`maxObjects` in the class definition above. In particular, be prepared for them to complain about the specification of 10 as an initial value for that variable. The ability to specify initial values for static `const` members (of integral type, e.g., `int`s, `char`s, enums, etc.) inside a class definition was added to C++ only relatively recently, so some compilers don't yet allow it. If your compilers are as-yet-unupdated, pacify them by declaring `maxObjects` to be an enumerator inside a private anonymous enum,

Click here to view code image

```
class Printer {
private:
  enum { maxObjects = 10 };    // within this class,
  ...                          // maxObjects is the
};                             // constant 10
```

or by initializing the constant static like a non-`const` static member:

Click here to view code image

```
class Printer {
private:
  static const size_t maxObjects;    // no initial value
given
```

```
    ...

  };

  // this goes in a single implementation file
  const size_t Printer::maxObjects = 10;
```

This latter approach has the same effect as the original code above, but explicitly specifying the initial value is easier for other programmers to understand. When your compilers support the specification of initial values for `const` static members in class definitions, you should take advantage of that capability.

An Object-Counting Base Class

Initialization of statics aside, the approach above works like the proverbial charm, but there is one aspect of it that continues to nag. If we had a lot of classes like `Printer` whose instantiations needed to be limited, we'd have to write this same code over and over, once per class. That would be mind-numbingly dull. Given a fancy-pants language like C++, it somehow seems we should be able to automate the process. Isn't there a way to encapsulate the notion of counting instances and bundle it into a class?

We can easily come up with a base class for counting object instances and have classes like `Printer` inherit from that, but it turns out we can do even better. We can actually come up with a way to encapsulate the whole counting kit and kaboodle, by which I mean not only the functions to manipulate the instance count, but also the instance count itself. (We'll see the need for a similar trick when we examine reference counting in Item 29.)

The counter in the `Printer` class is the static variable `numObjects`, so we need to move that variable into an instance-counting class. However, we also need to make sure that each class for which we're counting instances has a *separate* counter. Use of a counting class *template* lets us automatically generate the appropriate number of counters, because we can make the counter a static member of the classes generated from the template:

Click here to view code image

```
  template<class BeingCounted>
  class Counted {
```

```
public:
  class TooManyObjects{};       // for throwing exceptions

  static size_t objectCount() { return numObjects; }

protected:
  Counted();
  Counted(const Counted& rhs);

  ~Counted() { --numObjects; }

private:
  static size_t numObjects;
  static const size_t maxObjects;

  void init();                  // to avoid ctor code
};                              // duplication

template<class BeingCounted>
Counted<BeingCounted>::Counted()
{ init(); }

template<class BeingCounted>
Counted<BeingCounted>::Counted(const
Counted<BeingCounted>&)
{ init(); }

template<class BeingCounted>
void Counted<BeingCounted>::init()
{
  if (numObjects >= maxObjects) throw TooManyObjects();
  ++numObjects;
}
```

The classes generated from this template are designed to be used only as base classes, hence the protected constructors and destructor. Note the use of the private member function `init` to avoid duplicating the statements in the two `Counted` constructors.

We can now modify the `Printer` class to use the `Counted` template:

Click here to view code image

```
class Printer: private Counted<Printer> {
public:
  // pseudo-constructors
```

```
    static Printer * makePrinter();
    static Printer * makePrinter(const Printer& rhs);

    ~Printer();

    void submitJob(const PrintJob& job);
    void reset();
    void performSelfTest();
    ...

    using Counted<Printer>::objectCount;      // see below
    using Counted<Printer>::TooManyObjects;   // see below

  private:
    Printer();
    Printer(const Printer& rhs);
  };
```

The fact that `Printer` uses the `Counted` template to keep track of how many `Printer` objects exist is, frankly, nobody's business but the author of `Printer`'s. Such implementation details are best kept private, and that's why private inheritance is used here. The alternative would be to use public inheritance between `Printer` and `Counted<Printer>`, but then we'd be obliged to give the `Counted` classes a virtual destructor. (Otherwise we'd risk incorrect behavior if somebody deleted a `Printer` object through a `Counted<Printer>*` pointer.) As Item 24 makes clear, the presence of a virtual function in `Counted` would almost certainly affect the size and layout of objects of classes inheriting from `Counted`. We don't want to absorb that overhead, and the use of private inheritance lets us avoid it.

Quite properly, most of what `Counted` does is hidden from `Printer`'s clients, but those clients might reasonably want to find out how many `Printer` objects exist. The `Counted` template offers the `objectCount` function to provide this information, but that function becomes private in `Printer` due to our use of private inheritance. To restore the public accessibility of that function, we employ a `using` declaration:

Click here to view code image

```
  class Printer: private Counted<Printer> {
  public:
    ...
    using Counted<Printer>::objectCount;  // make this
```

```
function
                                        // public for
clients
  ...                                   // of Printer
};
```

This is perfectly legitimate, but if your compilers don't yet support namespaces, they won't allow it. If they don't, you can use the older access declaration syntax:

Click here to view code image

```
class Printer: private Counted<Printer> {
public:
  ...
  Counted<Printer>::objectCount;      // make objectCount
                                      // public in Printer
  ...

};
```

This more traditional syntax has the same meaning as the `using` declaration, but it's deprecated. The class `TooManyObjects` is handled in the same fashion as `objectCount`, because clients of `Printer` must have access to `TooManyObjects` if they are to be able to catch exceptions of that type.

When `Printer` inherits from `Counted<Printer>`, it can forget about counting objects. The class can be written as if somebody else were doing the counting for it, because somebody else (`Counted<Printer>`) is. A `Printer` constructor now looks like this:

```
Printer::Printer()
{
  proceed with normal object construction;
}
```

What's interesting here is not what you see, it's what you don't. No checking of the number of objects to see if the limit is about to be exceeded, no incrementing the number of objects in existence once the constructor is done. All that is now handled by the `Counted<Printer>` constructors, and because `Counted<Printer>` is a base class of `Printer`, we know that a `Counted<Printer>` constructor will always be called before a `Printer` constructor. If too many objects are created, a `Counted<Printer>` constructor throws an exception, and the `Printer` constructor won't even be invoked. Nifty, huh?

Nifty or not, there's one loose end that demands to be tied, and that's the mandatory definitions of the statics inside `Counted`. It's easy enough to take care of `numObjects` — we just put this in `Counted`'s implementation file:

Click here to view code image

```
template<class BeingCounted>                 // defines
numObjects
size_t Counted<BeingCounted>::numObjects; // and
automatically
                                             // initializes it
to 0
```

The situation with `maxObjects` is a bit trickier. To what value should we initialize this variable? If we want to allow up to 10 printers, we should initialize `Counted<Printer>::maxObjects` to 10. If, on the other hand, we want to allow up to 16 file descriptor objects, we should initialize `Counted<FileDescriptor>::maxObjects` to 16. What to do?

We take the easy way out: we do nothing. We provide no initialization at all for `maxObjects`. Instead, we require that *clients* of the class provide the appropriate initialization. The author of `Printer` must add this to an implementation file:

```
const size_t Counted<Printer>::maxObjects = 10;
```

Similarly, the author of `FileDescriptor` must add this:

```
const size_t Counted<FileDescriptor>::maxObjects = 16;
```

What will happen if these authors forget to provide a suitable definition for `maxObjects`? Simple: they'll get an error during linking, because `maxObjects` will be undefined. Provided we've adequately documented this requirement for clients of `Counted`, they can then say "Duh" to themselves and go back and add the requisite initialization.

Item 27: Requiring or prohibiting heap-based objects

Sometimes you want to arrange things so that objects of a particular type can commit suicide, i.e., can "`delete this`." Such an arrangement clearly requires that objects of that type be allocated on the heap. Other times you'll want to bask in the certainty that there can be no memory leaks for a particular class, because none of the objects could have been allocated on the heap. This might be the case if you are working on an embedded system, where memory leaks are especially troublesome and heap space is at a premium. Is it possible to produce code that requires or prohibits heap-based objects? Often it is, but it also turns out that the notion of being "on the heap" is more nebulous than you might think.

Requiring Heap-Based Objects

Let us begin with the prospect of limiting object creation to the heap. To enforce such a restriction, you've got to find a way to prevent clients from creating objects other than by calling `new`. This is easy to do. Non-heap objects are automatically constructed at their point of definition and automatically destructed at the end of their lifetime, so it suffices to simply make these implicit constructions and destructions illegal.

The straightforward way to make these calls illegal is to declare the constructors and the destructor `private`. This is overkill. There's no reason why they *both* need to be private. Better to make the destructor private and the constructors public. Then, in a process that should be familiar from Item 26, you can introduce a privileged pseudo-destructor function that has access to the real destructor. Clients then call the pseudo-destructor to destroy the objects they've created.

If, for example, we want to ensure that objects representing unlimited precision numbers are created only on the heap, we can do it like this:

Click here to view code image

```
class UPNumber {
public:
  UPNumber();
  UPNumber(int initValue);
  UPNumber(double initValue);
  UPNumber(const UPNumber& rhs);

  // pseudo-destructor (a const member function, because
  // even const objects may be destroyed)
  void destroy() const { delete this; }

  ...

private:
  ~UPNumber();
};
```

Clients would then program like this:

Click here to view code image

```
UPNumber n;                         // error! (legal here, but
                                    // illegal when n's dtor is
                                    // later implicitly invoked)

UPNumber *p = new UPNumber;         // fine

...

delete p;                           // error! attempt to call
                                    // private destructor

p->destroy();                       // fine
```

An alternative is to declare all the constructors private. The drawback to that idea is that a class often has many constructors, and the class's author must remember to declare each of them private. This includes the copy constructor, and it may include a default constructor, too, if these functions would otherwise be generated by compilers; compiler-generated functions are always public. As a result, it's easier to declare only the destructor private, because a class can have only one of those.

Restricting access to a class's destructor or its constructors prevents the creation of non-heap objects, but, in a story that is told in Item 26, it also prevents both inheritance and containment:

Click here to view code image

```
class UPNumber { ... };             // declares dtor or ctors
                                    // private

class NonNegativeUPNumber:
  public UPNumber { ... };          // error! dtor or ctors
                                    // won't compile
class Asset {
private:
  UPNumber value;
  ...                               // error! dtor or ctors
                                    // won't compile
};
```

Neither of these difficulties is insurmountable. The inheritance problem can be solved by making `UPNumber`'s destructor protected (while keeping its constructors public), and classes that need to contain objects of type `UPNumber` can be modified to contain *pointers* to `UPNumber` objects instead:

Click here to view code image

```
class UPNumber { ... };          // declares dtor protected

class NonNegativeUPNumber:
  public UPNumber { ... };       // now okay; derived
                                 // classes have access to
                                 // protected members

class Asset {
public:
  Asset(int initValue);
  ~Asset();
  ...

private:
  UPNumber *value;
};

Asset::Asset(int initValue)
: value(new UPNumber(initValue))     // fine
{ ... }

Asset::~Asset()
{ value->destroy(); }                // also fine
```

Determining Whether an Object is On The Heap

If we adopt this strategy, we must reexamine what it means to be "on the heap." Given the class definition sketched above, it's legal to define a non-heap `NonNegativeUPNumber` object:

```
NonNegativeUPNumber n;            // fine
```

Now, the `UPNumber` part of the `NonNegativeUPNumber` object `n` is not on the heap. Is that okay? The answer depends on the details of the class's design and implementation, but let us suppose it is *not* okay, that all `UPNumber` objects — even base class parts of more derived objects — *must* be on the heap. How can we enforce this restriction?

There is no easy way. It is not possible for a `UPNumber` constructor to determine whether it's being invoked as the base class part of a heap-based object. That is, there is no way for the `UPNumber` constructor to detect that the following contexts are different:

Click here to view code image

```
NonNegativeUPNumber *n1 =
  new NonNegativeUPNumber;      // on heap

NonNegativeUPNumber n2;         // not on heap
```

But perhaps you don't believe me. Perhaps you think you can play games with the interaction among the `new` operator, `operator new` and the constructor that the `new` operator calls (see Item 8). Perhaps you think you can outsmart them all by modifying `UPNumber` as follows:

Click here to view code image

```
class UPNumber {
public:
  // exception to throw if a non-heap object is created
  class HeapConstraintViolation {};

  static void * operator new(size_t size);

  UPNumber();
  ...

private:
  static bool onTheHeap;          // inside ctors, whether
                                  // the object being
  ...                             // constructed is on heap

};

// obligatory definition of class static
bool UPNumber::onTheHeap = false;

void *UPNumber::operator new(size_t size)
{
  onTheHeap = true;
  return ::operator new(size);
}
```

```
UPNumber::UPNumber()
{
  if (!onTheHeap) {
    throw HeapConstraintViolation();
  }

 proceed with normal construction here;

 onTheHeap = false;            // clear flag for next obj.
}
```

There's nothing deep going on here. The idea is to take advantage of the fact that when an object is allocated on the heap, `operator new` is called to allocate the raw memory, then a constructor is called to initialize an object in that memory. In particular, `operator new` sets `onTheHeap` to true, and each constructor checks `onTheHeap` to see if the raw memory of the object being constructed was allocated by `operator new`. If not, an exception of type `HeapConstraintViolation` is thrown. Otherwise, construction proceeds as usual, and when construction is finished, `onTheHeap` is set to false, thus resetting the default value for the next object to be constructed.

This is a nice enough idea, but it won't work. Consider this potential client code:

```
UPNumber *numberArray = new UPNumber[100];
```

The first problem is that the memory for the array is allocated by `operator new[]`, not `operator new`, but (provided your compilers support it) you can write the former function as easily as the latter. What is more troublesome is the fact that `numberArray` has 100 elements, so there will be 100 constructor calls. But there is only one call to allocate memory, so `onTheHeap` will be set to true for only the first of those 100 constructors. When the second constructor is called, an exception is thrown, and woe is you.

Even without arrays, this bit-setting business may fail. Consider this statement:

```
UPNumber *pn = new UPNumber(*new UPNumber);
```

Here we create two `UPNumber`s on the heap and make `pn` point to one of them; it's initialized with the value of the second one. This code has a resource leak, but let us ignore that in favor of an examination of what happens during execution of this expression:

```
new UPNumber(*new UPNumber)
```

This contains two calls to the `new` operator, hence two calls to `operator` new and two calls to `UPNumber` constructors (see Item 8). Programmers typically expect these function calls to be executed in this order,

1. Call `operator new` for first object (the leftmost one above)
2. Call constructor for first object
3. Call `operator new` for second object (the one used as an argument to the first `UPNumber`'s constructor)
4. Call constructor for second object

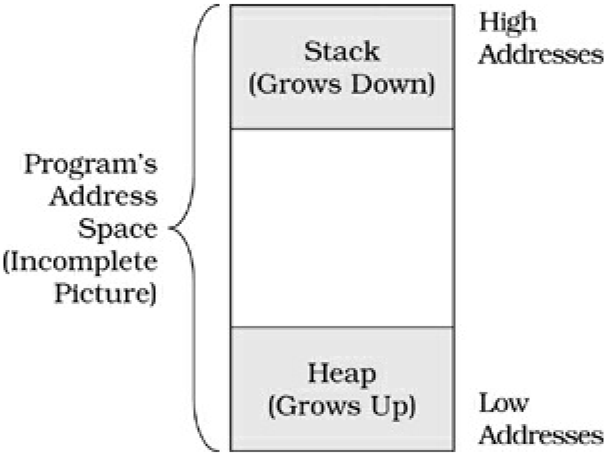
but the language makes no guarantee that this is how it will be done. Some compilers generate the function calls in this order instead:

1. Call `operator new` for first object
2. Call `operator new` for second object
3. Call constructor for second object
4. Call constructor for first object

There is nothing wrong with compilers that generate this kind of code, but the set-a-bit-in-`operator`-`new` trick fails with such compilers. That's because the bit set in steps 1 and 2 is cleared in step 3, thus making the object constructed in step 3 think it's not on the heap, even though it is.

These difficulties don't invalidate the basic idea of having each constructor check to see if `*this` is on the heap. Rather, they indicate that checking a bit set inside `operator new` (or `operator new[]`) is not a reliable way to determine this information. What we need is a better way to figure it out.

If you're desperate enough, you might be tempted to descend into the realm of the unportable. For example, you might decide to take advantage of the fact that on many systems, a program's address space is organized as a linear sequence of addresses, with the program's stack growing down from the top of the address space and the heap rising up from the bottom:



On systems that organize a program's memory in this way (many do, but many do not), you might think you could use the following function to determine whether a particular address is on the heap:

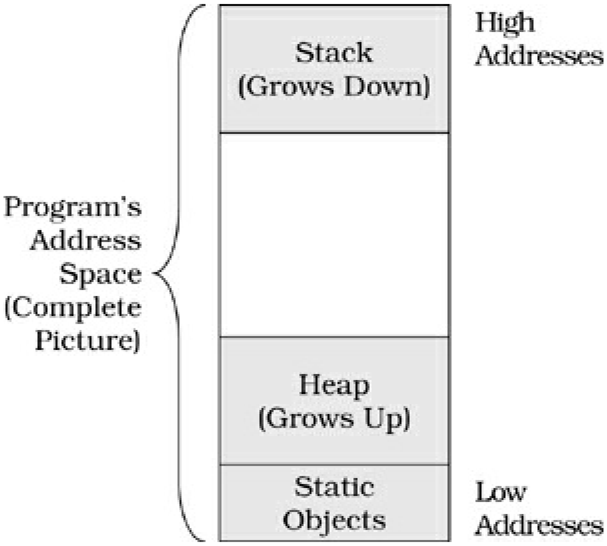
Click here to view code image

```
// incorrect attempt to determine whether an address
// is on the heap
bool onHeap(const void *address)
{
  char onTheStack;                // local stack variable

  return address < &onTheStack;
}
```

The thinking behind this function is interesting. Inside `onHeap`, `onTheStack` is a local variable. As such, it is, well, it's on the stack. When `onHeap` is called, its stack frame (i.e., its activation record) will be placed at the top of the program's stack, and because the stack grows down (toward lower addresses) in this architecture, the address of `onTheStack` must be less than the address of any other stack-based variable or object. If the parameter `address` is less than the location of `onTheStack`, it can't be on the stack, so it must be on the heap.

Such logic is fine, as far as it goes, but it doesn't go far enough. The fundamental problem is that there are *three* places where objects may be allocated, not two. Yes, the stack and the heap hold objects, but let us not forget about *static* objects. Static objects are those that are initialized only once during a program run. Static objects comprise not only those objects explicitly declared `static`, but also objects at global and namespace scope. Such objects have to go somewhere, and that somewhere is neither the stack nor the heap.

Where they go is system-dependent, but on many of the systems that have the stack and heap grow toward one another, they go below the heap. The earlier picture of memory organization, while telling the truth and nothing but the truth for many systems, failed to tell the whole truth for those systems. With static objects added to the picture, it looks like this:


Stack
(Grows Down)
High Addresses
Program's Address Space (Complete Picture)
Heap
(Grows Up)
Static Objects
Low Addresses

Suddenly it becomes clear why `onHeap` won't work, not even on systems where it's purported to: it fails to distinguish between heap objects and static objects:

[**Click here to view code image**]

```
void allocateSomeObjects()
{
  char *pc = new char;          // heap object: onHeap(pc)
                                // will return true
  char c;                       // stack object: onHeap(&c)
                                // will return false

  static char sc;               // static object:
onHeap(&sc)
                                // will return true
  ...
}
```

Now, you may be desperate for a way to tell heap objects from stack objects, and in your desperation you may be willing to strike a deal with the portability Devil, but are you so desperate that you'll strike a deal that fails to guarantee you the right answers? Surely not, so I know you'll reject this seductive but unreliable compare-the-addresses trick.

The sad fact is there's not only no portable way to determine whether an object is on the heap, there isn't even a semi-portable way that works most of the time.[†] If you absolutely, positively have to tell whether an address is on the heap, you're going to have to turn to unportable, implementation-dependent system calls, and that's that. That being the case, you're better off trying to redesign your software so you don't need to determine whether an object is on the heap in the first place.

If you find yourself obsessing over whether an object is on the heap, the likely cause is that you want to know if it's safe to invoke `delete` on it. Often such deletion will take the form of the infamous "`delete this`." Knowing whether it's safe to delete a pointer, however, is not the same as simply knowing whether that pointer points to something on the heap, because not all pointers to things on the heap can be safely `delete`d. Consider again an `Asset` object that contains a `UPNumber` object:

```
class Asset {
private:
  UPNumber value;
  ...
};

Asset *pa = new Asset;
```

Clearly `*pa` (including its member `value`) is on the heap. Equally clearly, it's not safe to invoke `delete` on a pointer to `pa->value`, because no such pointer was ever returned from `new`.

As luck would have it, it's easier to determine whether it's safe to delete a pointer than to determine whether a pointer points to something on the heap, because all we need to answer the former question is a collection of addresses that have been returned by `operator new`. Since we can write `operator new` ourselves, it's easy to construct such a collection. Here's how we might approach the problem:

Click here to view code image

```
void *operator new(size_t size)
{
    void *p = getMemory(size);         // call some function to
                                       // allocate memory and
                                       // handle out-of-memory
                                       // conditions

    add p to the collection of allocated addresses;

    return p;

  }

  void operator delete(void *ptr)
  {
    releaseMemory(ptr);                // return memory to
                                       // free store

    remove ptr from the collection of allocated addresses;
  }

  bool isSafeToDelete(const void *address)
  {
    return whether address is in collection of
    allocated addresses;
  }
```

This is about as simple as it gets. `operator new` adds entries to a collection of allocated addresses, `operator delete` removes entries, and `isSafeToDelete` does a lookup in the collection to see if a particular address is there. If the `operator new` and `operator delete` functions are at global scope, this should work for all types, even the built-ins.

In practice, three things are likely to dampen our enthusiasm for this design. The first is our extreme reluctance to define anything at global scope, especially functions with predefined meanings like `operator new` and `operator delete`. Knowing as we do that there is but one global scope and but a single version of `operator new` and `operator delete` with the "normal" signatures (i.e., sets of parameter types) within that scope, the last thing we want to do is seize those function signatures for ourselves. Doing so would render our software incompatible with any other software that also implements global versions of `operator new` and `operator delete` (such as many object-oriented database systems).

Our second consideration is one of efficiency: why burden all heap allocations with the bookkeeping overhead necessary to keep track of returned addresses if we don't need to?

Our final concern is pedestrian, but important. It turns out to be essentially impossible to implement `isSafeToDelete` so that it always works. The difficulty has to do with the fact that objects with multiple or virtual base classes have multiple addresses, so there's no guarantee that the address passed to `isSafeToDelete` is the same as the one returned from `operator new`, even if the object in question was allocated on the heap. For details, see Items 24 and 31.

What we'd like is the functionality provided by these functions without the concomitant pollution of the global namespace, the mandatory overhead, and the correctness problems. Fortunately, C++ gives us exactly what we need in the form of an abstract mixin base class.

An abstract base class is a base class that can't be instantiated, i.e., one with at least one pure virtual function. A mixin ("mix in") class is one that provides a single well-defined capability and is designed to be compatible with any other capabilities an inheriting class might provide. Such classes are nearly always abstract. We can therefore come up with an abstract mixin base class that offers derived classes the ability to determine whether a pointer was allocated from `operator new`. Here's such a class:

Click here to view code image

```
class HeapTracked {            // mixin class; keeps track of
public:                        // ptrs returned from op. new

  class MissingAddress{};      // exception class; see below

  virtual ~HeapTracked() = 0;

  static void *operator new(size_t size);
  static void operator delete(void *ptr);

  bool isOnHeap() const;

private:
  typedef const void* RawAddress;
  static list<RawAddress> addresses;
};
```

This class uses the `list` data structure that's part of the standard C++ library (see Item 35) to keep track of all pointers returned from `operator new`. That function allocates memory and adds entries to the list; `operator delete` deallocates memory and removes entries from the list; and `isOnHeap` returns whether an object's address is in the list.

Implementation of the `HeapTracked` class is simple, because the global `operator new` and `operator delete` functions are called to perform the real memory allocation and deallocation, and the `list` class has functions to make insertion, removal, and lookup single-statement operations. Here's the full implementation of `HeapTracked`:

Click here to view code image

```
// mandatory definition of static class member
list<RawAddress> HeapTracked::addresses;
// HeapTracked's destructor is pure virtual to make the
// class abstract. The destructor must still be
// defined, however, so we provide this empty definition.
HeapTracked::~HeapTracked() {}

void * HeapTracked::operator new(size_t size)
{
  void *memPtr = ::operator new(size);// get the memory

  addresses.push_front(memPtr);       // put its address at
                                      // the front of the
list
  return memPtr;
}

void HeapTracked::operator delete(void *ptr)
{
  // gracefully hande null pointers
  if (ptr == 0) return;

  // get an "iterator" that identifies the list
  // entry containing ptr; see Item 35 for details
  list<RawAddress>::iterator it =
    find(addresses.begin(), addresses.end(), ptr);

  if (it != addresses.end()) {   // if an entry was found
    addresses.erase(it);         // remove the entry
    ::operator delete(ptr);      // deallocate the memory
  } else {                       // otherwise
```

```
      throw MissingAddress();        // ptr wasn't allocated by
    }                                // op. new, so throw an
  }                                  // exception

  bool HeapTracked::isOnHeap() const
  {
    // get a pointer to the beginning of the memory
    // occupied by *this; see below for details
    const void *rawAddress = dynamic_cast<const void*>(this);

    // look up the pointer in the list of addresses
    // returned by operator new
    list<RawAddress>::iterator it =
      find(addresses.begin(), addresses.end(), rawAddress);

    return it != addresses.end();    // return whether it was
  }                                  // found
```

This code is straightforward, though it may not look that way if you are unfamiliar with the `list` class and the other components of the Standard Template Library. Item 35 explains everything, but the comments in the code above should be sufficient to explain what's happening in this example.

The only other thing that may confound you is this statement (in `isOnHeap`):

```
  const void *rawAddress = dynamic_cast<const void*>(this);
```

I mentioned earlier that writing the global function `isSafeToDelete` is complicated by the fact that objects with multiple or virtual base classes have several addresses. That problem plagues us in `isOnHeap`, too, but because `isOnHeap` applies only to `HeapTracked` objects, we can exploit a special feature of the `dynamic_cast` operator (see Item 2) to eliminate the problem. Simply put, `dynamic_cast`ing a pointer to `void*` (or `const void*` or `volatile void*` or, for those who can't get enough modifiers in their usual diet, `const volatile void*`) yields a pointer to the beginning of the memory for the object pointed to by the pointer. But `dynamic_cast` is applicable only to pointers to objects that have at least one virtual function. Our ill-fated `isSafeToDelete` function had to work with *any* type of pointer, so `dynamic_cast` wouldn't help it. `isOnHeap` is more selective (it tests only pointers to `HeapTracked` objects), so `dynamic_cast`ing `this` to `const void*` gives us a pointer to the beginning of the memory for the current object. That's the pointer `HeapTracked::operator new` must have returned if the memory for the current object was allocated by

`HeapTracked`::`operator new` in the first place. Provided your compilers support the `dynamic_cast` operator, this technique is completely portable.

Given this class, even BASIC programmers could add to a class the ability to track pointers to heap allocations. All they'd need to do is have the class inherit from `HeapTracked`. If, for example, we want to be able to determine whether a pointer to an `Asset` object points to a heap-based object, we'd modify `Asset`'s class definition to specify `HeapTracked` as a base class:

Click here to view code image

```
class Asset: public HeapTracked {
private:
  UPNumber value;
  ...

};
```

We could then query `Asset*` pointers as follows:

Click here to view code image

```
void inventoryAsset(const Asset *ap)
{
  if (ap->isOnHeap()) {
    ap is a heap-based asset - inventory it as such;
  }
  else {
    ap is a non-heap-based asset - record it that way;
  }
}
```

A disadvantage of a mixin class like `HeapTracked` is that it can't be used with the built-in types, because types like `int` and `char` can't inherit from anything. Still, the most common reason for wanting to use a class like `HeapTracked` is to determine whether it's okay to "`delete this`," and you'll never want to do that with a built-in type because such types have no `this` pointer.

Prohibiting Heap-Based Objects

Thus ends our examination of determining whether an object is on the heap. At the opposite end of the spectrum is *preventing* objects from being allocated on the heap. Here the outlook is a bit brighter. There are, as usual, three cases: objects that are directly instantiated, objects instantiated as base class parts of derived class objects, and objects embedded inside other objects. We'll consider each in turn.

Preventing clients from directly instantiating objects on the heap is easy, because such objects are always created by calls to `new` and you can make it impossible for clients to call `new`. Now, you can't affect the availability of the `new` operator (that's built into the language), but you can take advantage of the fact that the `new` operator always calls `operator new` (see Item 8), and that function is one you can declare yourself. In particular, it is one you can declare `private`. If, for example, you want to keep clients from creating `UPNumber` objects on the heap, you could do it this way:

Click here to view code image

```
class UPNumber {
private:
  static void *operator new(size_t size);
  static void operator delete(void *ptr);
  ...
};
```

Clients can now do only what they're supposed to be able to do:

Click here to view code image

```
UPNumber n1;                    // okay

static UPNumber n2;             // also okay

UPNumber *p = new UPNumber;     // error! attempt to call
                                // private operator new
```

It suffices to declare `operator new` private, but it looks strange to have `operator new` be private and `operator delete` be public, so unless there's a compelling reason to split up the pair, it's best to declare them in the same part of a class. If you'd like to prohibit heap-based arrays of `UPNumber` objects, too, you could declare `operator new[]` and `operator delete[]` (see Item 8) private as well.

Interestingly, declaring `operator new` private often also prevents `UPNumber` objects from being instantiated as base class parts of heap-based derived class objects. That's because `operator new` and `operator delete` are inherited, so if these functions aren't declared public in a derived class, that class inherits the private versions declared in its base(s):

Click here to view code image

```
class UPNumber { ... };          // as above

class NonNegativeUPNumber:       // assume this class
  public UPNumber {              // declares no operator new
  ...
};

NonNegativeUPNumber n1;          // okay

static NonNegativeUPNumber n2;   // also okay

NonNegativeUPNumber *p =         // error! attempt to call
  new NonNegativeUPNumber;       // private operator new
```

If the derived class declares an `operator new` of its own, that function will be called when allocating derived class objects on the heap, and a different way will have to be found to prevent `UPNumber` base class parts from winding up there. Similarly, the fact that `UPNumber`'s `operator new` is private has no effect on attempts to allocate objects containing `UPNumber` objects as members:

Click here to view code image

```
class Asset {
public:
  Asset(int initValue);
  ...

private:
  UPNumber value;
};

Asset *pa = new Asset(100);      // fine, calls
                                 // Asset::operator new or
                                 // ::operator new, not
                                 // UPNumber::operator new
```

For all practical purposes, this brings us back to where we were when we wanted to throw an exception in the `UPNumber` constructors if a `UPNumber` object was being constructed in memory that wasn't on the heap. This time, of course, we want to throw an exception if the object in question *is* on the heap. Just as there is no portable way to determine if an address is on the heap, however, there is no portable way to determine that it is not on the heap, so we're out of luck. This should be no surprise. After all, if we could tell when an address *is* on the heap, we could surely tell when an address is *not* on the heap. But we can't, so we can't. Oh well.

Item 28: Smart pointers

Smart pointers are objects that are designed to look, act, and feel like built-in pointers, but to offer greater functionality. They have a variety of applications, including resource management (see Items 9, 10, 25, and 31) and the automation of repetitive coding tasks (see Items 17 and 29).

When you use smart pointers in place of C++'s built-in pointers (i.e., *dumb* pointers), you gain control over the following aspects of pointer behavior:

- **Construction and destruction.** You determine what happens when a smart pointer is created and destroyed. It is common to give smart pointers a default value of 0 to avoid the headaches associated with uninitialized pointers. Some smart pointers are made responsible for deleting the object they point to when the last smart pointer pointing to the object is destroyed. This can go a long way toward eliminating resource leaks.
- **Copying and assignment.** You control what happens when a smart pointer is copied or is involved in an assignment. For some smart pointer types, the desired behavior is to automatically copy or make an assignment to what is pointed to, i.e., to perform a deep copy. For others, only the pointer itself should be copied or assigned. For still others, these operations should not be allowed at all. Regardless of what behavior you consider "right," the use of smart pointers lets you call the shots.
- **Dereferencing.** What should happen when a client refers to the object pointed to by a smart pointer? You get to decide. You could, for example, use smart pointers to help implement the lazy fetching strategy outlined in Item 17.

Smart pointers are generated from templates because, like built-in pointers, they must be strongly typed; the template parameter specifies the type of object pointed to. Most smart pointer templates look something like this:

Click here to view code image

```
template<class T>                    // template for smart
```

```
class SmartPtr {                  // pointer objects
public:
  SmartPtr(T* realPtr = 0);       // create a smart ptr to
an
                                  // obj given a dumb ptr
to
                                  // it; uninitialized ptrs
                                  // default to 0 (null)

  SmartPtr(const SmartPtr& rhs);  // copy a smart ptr

  ~SmartPtr();                    // destroy a smart ptr

  // make an assignment to a smart ptr
  SmartPtr& operator=(const SmartPtr& rhs);

  T* operator->() const;          // dereference a smart
ptr
                                  // to get at a member of
                                  // what it points to

  T& operator*() const;           // dereference a smart
ptr

private:
  T *pointee;                     // what the smart ptr
};                                // points to
```

The copy constructor and assignment operator are both shown public here. For smart pointer classes where copying and assignment are not allowed, they would typically be declared private. The two dereferencing operators are declared `const`, because dereferencing a pointer doesn't modify it (though it may lead to modification of what the pointer points to). Finally, each smart pointer-to-`T` object is implemented by containing a dumb pointer-to-`T` within it. It is this dumb pointer that does the actual pointing.

Before going into the details of smart pointer implementation, it's worth seeing how clients might use smart pointers. Consider a distributed system in which some objects are local and some are remote. Access to local objects is generally simpler and faster than access to remote objects, because remote access may require remote procedure calls or some other way of communicating with a distant machine.

For clients writing application code, the need to handle local and remote objects differently is a nuisance. It is more convenient to have all objects appear to be located in the same place. Smart pointers allow a library to offer this illusion:

Click here to view code image

```
template<class T>                // template for smart ptrs
class DBPtr {                    // to objects in a
public:                          // distributed DB

  DBPtr(T *realPtr = 0);         // create a smart ptr to a
                                 // DB object given a local
                                 // dumb pointer to it
  DBPtr(DataBaseID id);          // create a smart ptr to a
                                 // DB object given its
                                 // unique DB identifier

  ...                            // other smart ptr
};                               // functions as above

class Tuple {                    // class for database
public:                          // tuples
  ...
  void displayEditDialog();      // present a graphical
                                 // dialog box allowing a
                                 // user to edit the tuple

  bool isValid() const;          // return whether *this
};                               // passes validity check

// class template for making log entries whenever a T
// object is modified; see below for details
template<class T>
class LogEntry {
public:
  LogEntry(const T& objectToBeModified);
  ~LogEntry();
};

void editTuple(DBPtr<Tuple>& pt)
{
  LogEntry<Tuple> entry(*pt);    // make log entry for this
                                 // editing operation; see
                                 // below for details
```

```
    // repeatedly display edit dialog until valid values
    // are provided
    do {
      pt->displayEditDialog();
    } while (pt->isValid() == false);
  }
```

The tuple to be edited inside `editTuple` may be physically located on a remote machine, but the programmer writing `editTuple` need not be concerned with such matters; the smart pointer class hides that aspect of the system. As far as the programmer is concerned, all tuples are accessed through objects that, except for how they're declared, act just like run-of-the-mill built-in pointers.

Notice the use of a `LogEntry` object in `editTuple`. A more conventional design would have been to surround the call to `displayEditDialog` with calls to begin and end the log entry. In the approach shown here, the `LogEntry`'s constructor begins the log entry and its destructor ends the log entry. As Item 9 explains, using an object to begin and end logging is more robust in the face of exceptions than explicitly calling functions, so you should accustom yourself to using classes like `LogEntry`. Besides, it's easier to create a single `LogEntry` object than to add separate calls to start and stop an entry.

As you can see, using a smart pointer isn't much different from using the dumb pointer it replaces. That's testimony to the effectiveness of encapsulation. Clients of smart pointers are *supposed* to be able to treat them as dumb pointers. As we shall see, sometimes the substitution is more transparent than others.

Construction, Assignment, and Destruction of Smart Pointers

Construction of a smart pointer is usually straightforward: locate an object to point to (typically by using the smart pointer's constructor arguments), then make the smart pointer's internal dumb pointer point there. If no object can be located, set the internal pointer to 0 or signal an error (possibly by throwing an exception).

Implementing a smart pointer's copy constructor, assignment operator(s) and destructor is complicated somewhat by the issue of *ownership*. If a smart pointer *owns* the object it points to, it is responsible for deleting that object when it (the smart pointer) is destroyed. This assumes the object pointed to by the smart pointer is dynamically allocated. Such an assumption is common when working with smart pointers. (For ideas on how to make sure the assumption is true, see Item 27.)

Consider the `auto_ptr` template from the standard C++ library. As Item 9 explains, an `auto_ptr` object is a smart pointer that points to a heap-based object until it (the `auto_ptr`) is destroyed. When that happens, the `auto_ptr`'s destructor deletes the pointed-to object. The `auto_ptr` template might be implemented like this:

Click here to view code image

```
template<class T>
class auto_ptr {
public:
  auto_ptr(T *ptr = 0): pointee(ptr) {}
  ~auto_ptr() { delete pointee; }
  ...

private:
  T *pointee;
};
```

This works fine provided only one `auto_ptr` owns an object. But what should happen when an `auto_ptr` is copied or assigned?

Click here to view code image

```
auto_ptr<TreeNode> ptn1(new TreeNode);

auto_ptr<TreeNode> ptn2 = ptn1;     // call to copy ctor;
                                    // what should happen?
auto_ptr<TreeNode> ptn3;

ptn3 = ptn2;                        // call to operator=;
                                    // what should happen?
```

If we just copied the internal dumb pointer, we'd end up with two `auto_ptr`s pointing to the same object. This would lead to grief, because each `auto_ptr` would delete what it pointed to when the `auto_ptr` was destroyed. That would mean we'd delete an object more than once. The results of such double-deletes are undefined (and are frequently disastrous).

An alternative would be to create a new copy of what was pointed to by calling `new`. That would guarantee we didn't have too many `auto_ptr`s pointing to a single object, but it might engender an unacceptable performance hit for the creation (and later destruction) of the new object. Furthermore, we wouldn't necessarily know what type of object to create, because an `auto_ptr<T>` object need not point to an object of type `T`; it might point to an object of a type *derived* from `T`. Virtual constructors (see Item 25) can help solve this problem, but it seems inappropriate to require their use in a general-purpose class like `auto_ptr`.

The problems would vanish if `auto_ptr` prohibited copying and assignment, but a more flexible solution was adopted for the `auto_ptr` classes: object ownership is *transferred* when an `auto_ptr` is copied or assigned:

Click here to view code image

```
template<class T>
class auto_ptr {
public:
  ...

  auto_ptr(auto_ptr<T>& rhs);        // copy constructor

  auto_ptr<T>&                       // assignment
  operator=(auto_ptr<T>& rhs);       // operator

  ...
};

template<class T>
auto_ptr<T>::auto_ptr(auto_ptr<T>& rhs)
{
  pointee = rhs.pointee;        // transfer ownership of
                                // *pointee to *this

  rhs.pointee = 0;              // rhs no longer owns
}                               // anything
template<class T>
auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs)
{
  if (this == &rhs)             // do nothing if this
    return *this;               // object is being assigned
                                // to itself

  delete pointee;               // delete currently owned
                                // object
    pointee = rhs.pointee;         // transfer ownership of
    rhs.pointee = 0;               // *pointee from rhs to
  *this

    return *this;
  }
```

Notice that the assignment operator must delete the object it owns before assuming ownership of a new object. If it failed to do this, the object would never be deleted. Remember, nobody but the `auto_ptr` object owns the object the `auto_ptr` points to.

Because object ownership is transferred when `auto_ptr`'s copy constructor is called, passing `auto_ptr`s by value is often a *very* bad idea. Here's why:

Click here to view code image

```
  // this function will often lead to disaster
  void printTreeNode(ostream& s, auto_ptr<TreeNode> p)
  { s << *p; }

  int main()
  {
    auto_ptr<TreeNode> ptn(new TreeNode);

    ...

    printTreeNode(cout, ptn);     // pass auto_ptr by value

    ...

  }
```

When `printTreeNode`'s parameter `p` is initialized (by calling `auto_ptr`'s copy constructor), ownership of the object pointed to by `ptn` is transferred to `p`. When `printTreeNode` finishes executing, `p` goes out of scope and its destructor deletes what it points to (which is what `ptn` used to point to). `ptn`, however, no longer points to anything (its underlying dumb pointer is null), so just about any attempt to use it after the call to `printTreeNode` will yield undefined behavior. Passing `auto_ptr`s by value, then, is something to be done only if you're *sure* you want to transfer ownership of an object to a (transient) function parameter. Only rarely will you want to do this.

This doesn't mean you can't pass `auto_ptr`s as parameters, it just means that pass-by-value is not the way to do it. Pass-by-reference-to-`const` is:

Click here to view code image

```
// this function behaves much more intuitively
void printTreeNode(ostream& s,
                   const auto_ptr<TreeNode>& p)
{ s << *p; }
```

In this function, `p` is a reference, not an object, so no constructor is called to initialize `p`. When `ptn` is passed to this version of `printTreeNode`, it retains ownership of the object it points to, and `ptn` can safely be used after the call to `printTreeNode`. Thus, passing `auto_ptr`s by reference-to-`const` avoids the hazards arising from pass-by-value.

The notion of transferring ownership from one smart pointer to another during copying and assignment is interesting, but you may have been at least as interested in the unconventional declarations of the copy constructor and assignment operator. These functions normally take `const` parameters, but above they do not. In fact, the code above *changes* these parameters during the copy or the assignment. In other words, `auto_ptr` objects are modified if they are copied or are the source of an assignment!

Yes, that's exactly what's happening. Isn't it nice that C++ is flexible enough to let you do this? If the language required that copy constructors and assignment operators take `const` parameters, you'd probably have to cast away the parameters' `const`ness or play other games to implement ownership transferral. Instead, you get to say exactly what you want to say: when an object is copied or is the source of an assignment, that object is changed. This may not seem intuitive, but it's simple, direct, and, in this case, accurate.

If you find this examination of `auto_ptr` member functions interesting, you may wish to see a complete implementation. You'll find one on pages 291-294B, where you'll also see that the `auto_ptr` template in the standard C++ library has copy constructors and assignment operators that are more flexible than those described here. In the standard `auto_ptr` template, those functions are member function *templates*, not just member functions. (Member function templates are described later in this Item.)

A smart pointer's destructor often looks like this:

Click here to view code image

```
template<class T>
SmartPtr<T>::~SmartPtr()
{
  if (*this owns *pointee) {
    delete pointee;
  }
}
```

Sometimes there is no need for the test. An `auto_ptr` always owns what it points to, for example. At other times the test is a bit more complicated. A smart pointer that employs reference counting (see Item 29) must adjust a reference count before determining whether it has the right to delete what it points to. Of course, some smart pointers are like dumb pointers: they have no effect on the object they point to when they themselves are destroyed.

Implementing the Dereferencing Operators

Let us now turn our attention to the very heart of smart pointers, the `operator*` and `operator->` functions. The former returns the object pointed to. Conceptually, this is simple:

Click here to view code image

```
template<class T>
T& SmartPtr<T>::operator*() const
{
  perform "smart pointer" processing;

  return *pointee;
}
```

First the function does whatever processing is needed to initialize or otherwise make `pointee` valid. For example, if lazy fetching is being used (see Item 17), the function may have to conjure up a new object for `pointee` to point to. Once `pointee` is valid, the `operator*` function just returns a reference to the pointed-to object.

Note that the return type is a *reference*. It would be disastrous to return an *object* instead, though compilers will let you do it. Bear in mind that `pointee` need not point to an object of type `T`; it may point to an object of a class *derived* from `T`. If that is the case and your `operator*` function returns a `T` object instead of a reference to the actual derived class object, your function will return an object of the wrong type! (This is the *slicing problem* — see Item 13.) Virtual functions invoked on the object returned from your star-crossed `operator*` will not invoke the function corresponding to the dynamic type of the pointed-to object. In essence, your smart pointer will not properly support virtual functions, and how smart is a pointer like that? Besides, returning a reference is more efficient anyway, because there is no need to construct a temporary object (see Item 19). This is one of those happy occasions when correctness and efficiency go hand in hand.

If you're the kind who likes to worry, you may wonder what you should do if somebody invokes `operator*` on a null smart pointer, i.e., one whose embedded dumb pointer is null. Relax. You can do anything you want. The result of dereferencing a null pointer is undefined, so there is no "wrong" behavior. Wanna throw an exception? Go ahead, throw it. Wanna call `abort` (possibly by having an `assert` call fail)? Fine, call it. Wanna walk through memory setting every byte to your birth date modulo 256? That's okay, too. It's not nice, but as far as the language is concerned, you are completely unfettered.

The story with `operator->` is similar to that for `operator*`, but before examining `operator->`, let us remind ourselves of the unusual meaning of a call to this function. Consider again the `editTuple` function that uses a smart pointer-to-`Tuple` object:

Click here to view code image

```
void editTuple(DBPtr<Tuple>& pt)
{
  LogEntry<Tuple> entry(*pt);

  do {
    pt->displayEditDialog();
  } while (pt->isValid() == false);
}
```

The statement

```
pt->displayEditDialog();
```

is interpreted by compilers as:

```
(pt.operator->())->displayEditDialog();
```

That means that whatever `operator->` returns, it must be legal to apply the member-selection operator (`->`) to it. There are thus only two things `operator->` can return: a dumb pointer to an object or another smart pointer object. Most of the time, you'll want to return an ordinary dumb pointer. In those cases, you implement `operator->` as follows:

Click here to view code image

```
template<class T>
T* SmartPtr<T>::operator->() const
{
  perform "smart pointer" processing;

  return pointee;
}
```

This will work fine. Because this function returns a pointer, virtual function calls via `operator->` will behave the way they're supposed to. For many applications, this is all you need to know about smart pointers. The reference-counting code of Item 29, for example, draws on no more functionality than we've discussed here. If you want to push your smart pointers further, however, you must know more about dumb pointer behavior and how smart pointers can and cannot emulate it. If your motto is "Most people stop at the Z — but not me!", the material that follows is for you.

Testing Smart Pointers for Nullness

With the functions we have discussed so far, we can create, destroy, copy, assign, and dereference smart pointers. One of the things we cannot do, however, is find out if a smart pointer is null:

Click here to view code image

```
SmartPtr<TreeNode> ptn;

...

if (ptn == 0) ...              // error!

if (ptn) ...                   // error!

if (!ptn) ...                  // error!
```

This is a serious limitation.

It would be easy to add an `isNull` member function to our smart pointer classes, but that wouldn't address the problem that smart pointers don't act like dumb pointers when testing for nullness. A different approach is to provide an implicit conversion operator that allows the tests above to compile. The conversion traditionally employed for this purpose is to `void*`:

Click here to view code image

```
template<class T>
class SmartPtr {
public:
  ...
  operator void*();             // returns 0 if the smart
  ...                           // ptr is null, nonzero
};                              // otherwise

SmartPtr<TreeNode> ptn;

...

if (ptn == 0) ...               // now fine

if (ptn) ...                    // also fine

if (!ptn) ...                   // fine
```

This is similar to a conversion provided by the iostream classes, and it explains why it's possible to write code like this:

Click here to view code image

```
ifstream inputFile("datafile.dat");
if (inputFile) ...             // test to see if inputFile
                               // was successfully
                               // opened
```

Like all type conversion functions, this one has the drawback of letting function calls succeed that most programmers would expect to fail (see Item 5). In particular, it allows comparisons of smart pointers of completely different types:

Click here to view code image

```
SmartPtr<Apple> pa;
SmartPtr<Orange> po;
```

```
...

if (pa == po) ...              // this compiles!
```

Even if there is no `operator==` taking a `SmartPtr<Apple>` and a `SmartPtr<Orange>`, this compiles, because both smart pointers can be implicitly converted into `void*` pointers, and there is a built-in comparison function for built-in pointers. This kind of behavior makes implicit conversion functions dangerous. (Again, see Item 5, and keep seeing it over and over until you can see it in the dark.)

There are variations on the conversion-to-`void*` motif. Some designers advocate conversion to `const void*`, others embrace conversion to `bool`. Neither of these variations eliminates the problem of allowing mixed-type comparisons.

There is a middle ground that allows you to offer a reasonable syntactic form for testing for nullness while minimizing the chances of accidentally comparing smart pointers of different types. It is to overload `operator!` for your smart pointer classes so that `operator!` returns `true` if and only if the smart pointer on which it's invoked is null:

Click here to view code image

```
template<class T>
class SmartPtr {
public:
  ...
  bool operator!() const;      // returns true if and only
  ...                          // if the smart ptr is null

};
```

This lets your clients program like this,

Click here to view code image

```
SmartPtr<TreeNode> ptn;

...

if (!ptn) {                    // fine
  ...                          // ptn is null
}
else {
    ...                          // ptn is not null
  }
```

but not like this:

Click here to view code image

```
  if (ptn == 0) ...              // still an error

  if (ptn) ...                   // also an error
```

The only risk for mixed-type comparisons is statements such as these:

Click here to view code image

```
  SmartPtr<Apple> pa;
  SmartPtr<Orange> po;

  ...

  if (!pa == !po) ...            // alas, this compiles
```

Fortunately, programmers don't write code like this very often. Interestingly, iostream library implementations provide an `operator!` in addition to the implicit conversion to `void*`, but these two functions typically test for slightly different stream states. (In the C++ library standard (see Item 35), the implicit conversion to `void*` has been replaced by an implicit conversion to `bool`, and `operator bool` always returns the negation of `operator!`.)

Converting Smart Pointers to Dumb Pointers

Sometimes you'd like to add smart pointers to an application or library that already uses dumb pointers. For example, your distributed database system may not originally have been distributed, so you may have some old library functions that aren't designed to use smart pointers:

Click here to view code image

```
  class Tuple { ... };           // as before

  void normalize(Tuple *pt);     // put *pt into canonical
                                 // form; note use of dumb
                                 // pointer
```

Consider what will happen if you try to call `normalize` with a smart pointer-to-`Tuple`:

[Click here to view code image](#)

```
DBPtr<Tuple> pt;

...

normalize(pt);                    // error!
```

The call will fail to compile, because there is no way to convert a `DBPtr<Tuple>` to a `Tuple*`. You can make it work by doing this,

```
normalize(&*pt);                  // gross, but legal
```

but I hope you'll agree this is repugnant.

The call can be made to succeed by adding to the smart pointer-to-T template an implicit conversion operator to a dumb pointer-to-T:

[Click here to view code image](#)

```
template<class T>                 // as before
class DBPtr {
public:
  ...
  operator T*() { return pointee; }
  ...
};

DBPtr<Tuple> pt;

...

normalize(pt);                    // this now works
```

Addition of this function also eliminates the problem of testing for nullness:

[Click here to view code image](#)

```
if (pt == 0) ...                  // fine, converts pt to a
                                  // Tuple*

if (pt) ...                       // ditto

if (!pt) ...                      // ditto (reprise)
```

However, there is a dark side to such conversion functions. (There almost always is. Have you been seeing [Item 5](#)?) They make it easy for clients to program directly with dumb pointers, thus bypassing the smarts your pointer-like objects are designed to provide:

Click here to view code image

```
void processTuple(DBPtr<Tuple>& pt)
{
  Tuple *rawTuplePtr = pt;      // converts DBPtr<Tuple> to
                                // Tuple*

  use rawTuplePtr to modify the tuple;

}
```

Usually, the "smart" behavior provided by a smart pointer is an essential component of your design, so allowing clients to use dumb pointers typically leads to disaster. For example, if `DBPtr` implements the reference-counting strategy of Item 29, allowing clients to manipulate dumb pointers directly will almost certainly lead to bookkeeping errors that corrupt the reference-counting data structures.

Even if you provide an implicit conversion operator to go from a smart pointer to the dumb pointer it's built on, your smart pointer will never be truly interchangeable with the dumb pointer. That's because the conversion from a smart pointer to a dumb pointer is a user-defined conversion, and compilers are forbidden from applying more than one such conversion at a time. For example, suppose you have a class representing all the clients who have accessed a particular tuple:

Click here to view code image

```
class TupleAccessors {
public:
  TupleAccessors(const Tuple *pt);    // pt identifies the
  ...                                 // tuple whose
accessors
};                                    // we care about
```

As usual, `TupleAccessors`' single-argument constructor also acts as a type-conversion operator from `Tuple*` to `TupleAccessors` (see Item 5). Now consider a function for merging the information in two `TupleAccessors` objects:

Click here to view code image

```
TupleAccessors merge(const TupleAccessors& ta1,
                     const TupleAccessors& ta2);
```

Because a `Tuple*` may be implicitly converted to a `TupleAccessors`, calling `merge` with two dumb `Tuple*` pointers is fine:

Click here to view code image

```
Tuple *pt1, *pt2;

...

merge(pt1, pt2);          // fine, both pointers are converted
                          // to TupleAccessors objects
```

The corresponding call with smart `DBPtr<Tuple>` pointers, however, fails to compile:

Click here to view code image

```
DBPtr<Tuple> pt1, pt2;

...

merge(pt1, pt2);          // error! No way to convert pt1 and
                          // pt2 to TupleAccessors objects
```

That's because a conversion from `DBPtr<Tuple>` to `TupleAccessors` calls for *two* user-defined conversions (one from `DBPtr<Tuple>` to `Tuple*` and one from `Tuple*` to `TupleAccessors`), and such sequences of conversions are prohibited by the language.

Smart pointer classes that provide an implicit conversion to a dumb pointer open the door to a particularly nasty bug. Consider this code:

Click here to view code image

```
DBPtr<Tuple> pt = new Tuple;

...

delete pt;
```

This should not compile. After all, `pt` is not a pointer, it's an object, and you can't delete an object. Only pointers can be deleted, right?

Right. But remember from Item 5 that compilers use implicit type conversions to make function calls succeed whenever they can, and recall from Item 8 that use of the `delete` operator leads to calls to a destructor and to `operator delete`, both of which are functions. Compilers want these function calls to succeed, so in the `delete` statement above, they implicitly convert `pt` to a `Tuple*`, then they delete that. This will almost certainly break your program.

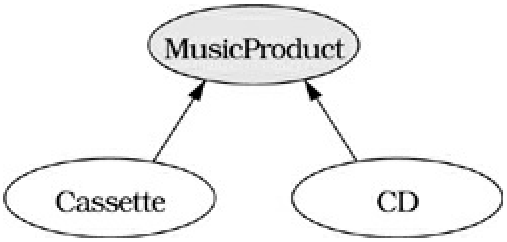
If `pt` owns the object it points to, that object is now deleted twice, once at the point where `delete` is called, a second time when `pt`'s destructor is invoked. If `pt` doesn't own the object, somebody else does. That somebody may be the person who deleted `pt`, in which case all is well. If, however, the owner of the object pointed to by `pt` is not the person who deleted `pt`, we can expect the rightful owner to delete that object again later. The first and last of these scenarios leads to an object being deleted twice, and deleting an object more than once yields undefined behavior.

This bug is especially pernicious because the whole idea behind smart pointers is to make them look and feel as much like dumb pointers as possible. The closer you get to this ideal, the more likely your clients are to forget they are using smart pointers. If they do, who can blame them if they continue to think that in order to avoid resource leaks, they must call `delete` if they called `new`?

The bottom line is simple: don't provide implicit conversion operators to dumb pointers unless there is a compelling reason to do so.

Smart Pointers and Inheritance-Based Type Conversions

Suppose we have a public inheritance hierarchy modeling consumer products for storing music:



[Click here to view code image](#)

```
class MusicProduct {
public:
  MusicProduct(const string& title);
  virtual void play() const = 0;
  virtual void displayTitle() const = 0;
  ...
};
class Cassette: public MusicProduct {
public:
  Cassette(const string& title);
  virtual void play() const;
  virtual void displayTitle() const;
  ...
};

class CD: public MusicProduct {
public:
  CD(const string& title);
  virtual void play() const;
  virtual void displayTitle() const;
  ...
};
```

Further suppose we have a function that, given a `MusicProduct` object, displays the title of the product and then plays it:

Click here to view code image

```
void displayAndPlay(const MusicProduct* pmp, int numTimes)
{
  for (int i = 1; i <= numTimes; ++i) {
    pmp->displayTitle();
    pmp->play();
  }
}
```

Such a function might be used like this:

Click here to view code image

```
Cassette *funMusic = new Cassette("Alapalooza");
CD *nightmareMusic = new CD("Disco Hits of the 70s");

displayAndPlay(funMusic, 10);
displayAndPlay(nightmareMusic, 0);
```

There are no surprises here, but look what happens if we replace the dumb pointers with their allegedly smart counterparts:

Click here to view code image

```
void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int numTimes);
```

```
SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));
SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the
70s"));

displayAndPlay(funMusic, 10);          // error!
displayAndPlay(nightmareMusic, 0);     // error!
```

If smart pointers are so brainy, why won't these compile?

They won't compile because there is no conversion from a `SmartPtr<CD>` or a `SmartPtr<Cassette>` to a `SmartPtr<MusicProduct>`. As far as compilers are concerned, these are three separate classes — they have no relationship to one another. Why should compilers think otherwise? After all, it's not like `SmartPtr<CD>` or `SmartPtr<Cassette>` inherits from `SmartPtr<MusicProduct>`. With no inheritance relationship between these classes, we can hardly expect compilers to run around converting objects of one type to objects of other types.

Fortunately, there is a way to get around this limitation, and the idea (if not the practice) is simple: give each smart pointer class an implicit type conversion operator (see Item 5) for each smart pointer class to which it should be implicitly convertible. For example, in the music hierarchy, you'd add an `operator SmartPtr<MusicProduct>` to the smart pointer classes for `Cassette` and `CD`:

Click here to view code image

```
class SmartPtr<Cassette> {
public:
  operator SmartPtr<MusicProduct>()
  { return SmartPtr<MusicProduct>(pointee); }

  ...

private:
  Cassette *pointee;
};

class SmartPtr<CD> {
public:
  operator SmartPtr<MusicProduct>()
  { return SmartPtr<MusicProduct>(pointee); }
```

```
    ...

  private:
    CD *pointee;
  };
```

The drawbacks to this approach are twofold. First, you must manually specialize the `SmartPtr` class instantiations so you can add the necessary implicit type conversion operators, but that pretty much defeats the purpose of templates. Second, you may have to add many such conversion operators, because your pointed-to object may be deep in an inheritance hierarchy, and you must provide a conversion operator for *each* base class from which that object directly or indirectly inherits. (If you think you can get around this by providing only an implicit type conversion operator for each direct base class, think again. Because compilers are prohibited from employing more than one user-defined type conversion function at a time, they can't convert a smart pointer-to-`T` to a smart pointer-to-indirect-base-class-of-`T` unless they can do it in a single step.)

It would be quite the time-saver if you could somehow get compilers to write all these implicit type conversion functions for you. Thanks to a recent language extension, you can. The extension in question is the ability to declare (nonvirtual) *member function templates* (usually just called *member templates*), and you use it to generate smart pointer conversion functions like this:

Click here to view code image

```
  template<class T>                // template class for smart
  class SmartPtr {                 // pointers-to-T objects
  public:
    SmartPtr(T* realPtr = 0);

    T* operator->() const;
    T& operator*() const;

    template<class newType>        // template function for
    operator SmartPtr<newType>()   // implicit conversion ops.
    {
      return SmartPtr<newType>(pointee);
    }

    ...
```

```
};
```

Now hold on to your headlights, this isn't magic — but it's close. It works as follows. (I'll give a specific example in a moment, so don't despair if the remainder of this paragraph reads like so much gobbledygook. After you've seen the example, it'll make more sense, I promise.) Suppose a compiler has a smart pointer-to-`T` object, and it's faced with the need to convert that object into a smart pointer-to-base-class-of-`T`. The compiler checks the class definition for `SmartPtr<T>` to see if the requisite conversion operator is declared, but it is not. (It can't be: no conversion operators are declared in the template above.) The compiler then checks to see if there's a member function template it can instantiate that would let it perform the conversion it's looking for. It finds such a template (the one taking the formal type parameter `newType`), so it instantiates the template with `newType` bound to the base class of `T` that's the target of the conversion. At that point, the only question is whether the code for the instantiated member function will compile. In order for it to compile, it must be legal to pass the (dumb) pointer `pointee` to the constructor for the smart pointer-to-base-of-`T`. `pointee` is of type `T`, so it is certainly legal to convert it into a pointer to its (public or protected) base classes. Hence, the code for the type conversion operator will compile, and the implicit conversion from smart pointer-to-`T` to smart pointer-to-base-of-`T` will succeed.

An example will help. Let us return to the music hierarchy of CDs, cassettes, and music products. We saw earlier that the following code wouldn't compile, because there was no way for compilers to convert the smart pointers to CDs or cassettes into smart pointers to music products:

Click here to view code image

```
void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int howMany);

SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));
SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the
70s"));

displayAndPlay(funMusic, 10);         // used to be an error
displayAndPlay(nightmareMusic, 0);    // used to be an error
```

With the revised smart pointer class containing the member function template for implicit type conversion operators, this code will succeed. To see why, look at this call:

```
displayAndPlay(funMusic, 10);
```

The object `funMusic` is of type `SmartPtr<Cassette>`. The function `displayAndPlay` expects a `SmartPtr<MusicProduct>` object. Compilers detect the type mismatch and seek a way to convert `funMusic` into a `SmartPtr<MusicProduct>` object. They look for a single-argument constructor (see Item 5) in the `SmartPtr<MusicProduct>` class that takes a `SmartPtr<Cassette>`, but they find none. They look for an implicit type conversion operator in the `SmartPtr<Cassette>` class that yields a `SmartPtr<MusicProduct>` class, but that search also fails. They then look for a member function template they can instantiate to yield one of these functions. They discover that the template inside `SmartPtr<Cassette>`, when instantiated with `newType` bound to `MusicProduct`, generates the necessary function. They instantiate the function, yielding the following code:

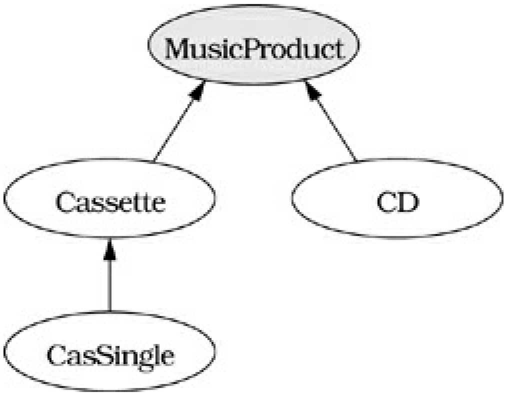
Click here to view code image

```
SmartPtr<Cassette>::operator SmartPtr<MusicProduct>()
{
  return SmartPtr<MusicProduct>(pointee);
}
```

Will this compile? For all intents and purposes, nothing is happening here except the calling of the `SmartPtr<MusicProduct>` constructor with `pointee` as its argument, so the real question is whether one can construct a `SmartPtr<MusicProduct>` object with a `Cassette*` pointer. The `SmartPtr<MusicProduct>` constructor expects a `MusicProduct*` pointer, but now we're on the familiar ground of conversions between dumb pointer types, and it's clear that `Cassette*` can be passed in where a `MusicProduct*` is expected. The construction of the `SmartPtr<MusicProduct>` is therefore successful, and the conversion of the `SmartPtr<Cassette>` to `SmartPtr<MusicProduct>` is equally successful. *Voilà!* Implicit conversion of smart pointer types. What could be simpler?

Furthermore, what could be more powerful? Don't be misled by this example into assuming that this works only for pointer conversions up an inheritance hierarchy. The method shown succeeds for *any* legal implicit conversion between pointer types. If you've got a dumb pointer type `T1*` and another dumb pointer type `T2*`, you can implicitly convert a smart pointer-to-`T1` to a smart pointer-to-`T2` if and only if you can implicitly convert a `T1*` to a `T2*`.

This technique gives you exactly the behavior you want — almost. Suppose we augment our `MusicProduct` hierarchy with a new class, `CasSingle`, for representing cassette singles. The revised hierarchy looks like this:



Now consider this code:

Click here to view code image

```
template<class T>           // as above, including member
tem-
class SmartPtr { ... };     // plate for conversion
operators

void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int howMany);

void displayAndPlay(const SmartPtr<Cassette>& pc,
                    int howMany);

SmartPtr<CasSingle> dumbMusic(new CasSingle("Achy Breaky
Heart"));

displayAndPlay(dumbMusic, 1);      // error!
```

In this example, `displayAndPlay` is overloaded, with one function taking a `SmartPtr<MusicProduct>` object and the other taking a `SmartPtr<Cassette>` object. When we invoke `displayAndPlay` with a `SmartPtr<CasSingle>`, we expect the `SmartPtr<Cassette>` function to be chosen, because `CasSingle` inherits directly from `Cassette` and only indirectly from `MusicProduct`. Certainly that's how it would work with dumb pointers. Alas, our smart pointers aren't that smart. They employ member functions as conversion operators, and as far as C++ compilers are concerned, all calls to conversion functions are equally good.

As a result, the call to `displayAndPlay` is ambiguous, because the conversion from `SmartPtr<CasSingle>` to `SmartPtr<Cassette>` is no better than the conversion to `SmartPtr<MusicProduct>`.

Implementing smart pointer conversions through member templates has two additional drawbacks. First, support for member templates is rare, so this technique is currently anything but portable. In the future, that will change, but nobody knows just how far in the future that will be. Second, the mechanics of why this works are far from transparent, relying as they do on a detailed understanding of argument-matching rules for function calls, implicit type conversion functions, implicit instantiation of template functions, and the existence of member function templates. Pity the poor programmer who has never seen this trick before and is then asked to maintain or enhance code that relies on it. The technique is clever, that's for sure, but too much cleverness can be a dangerous thing.

Let's stop beating around the bush. What we really want to know is how we can make smart pointer classes behave just like dumb pointers for purposes of inheritance-based type conversions. The answer is simple: we can't. As Daniel Edelson has noted, smart pointers are smart, but they're not pointers. The best we can do is to use member templates to generate conversion functions, then use casts (see Item 2) in those cases where ambiguity results. This isn't a perfect state of affairs, but it's pretty good, and having to cast away ambiguity in a few cases is a small price to pay for the sophisticated functionality smart pointers can provide.

Smart Pointers and const

Recall that for dumb pointers, `const` can refer to the thing pointed to, to the pointer itself, or both:

Click here to view code image

```
CD goodCD("Flood");

const CD *p;                     // p is a non-const pointer
                                 // to a const CD object

CD * const p = &goodCD;          // p is a const pointer to
                                 // a non-const CD object;
                                 // because p is const, it
```

```
                                // must be initialized

const CD * const p = &goodCD;   // p is a const pointer to
                                // a const CD object
```

Naturally, we'd like to have the same flexibility with smart pointers. Unfortunately, there's only one place to put the `const`, and there it applies to the pointer, not to the object pointed to:

Click here to view code image

```
const SmartPtr<CD> p =          // p is a const smart ptr
  &goodCD;                      // to a non-const CD object
```

This seems simple enough to remedy — just create a smart pointer to a *`const`* `CD`:

Click here to view code image

```
SmartPtr<const CD> p =          // p is a non-const smart
ptr
  &goodCD;                      // to a const CD object
```

Now we can create the four combinations of `const` and non-`const` objects and pointers we seek:

Click here to view code image

```
SmartPtr<CD> p;                       // non-const object,
                                      // non-const pointer

SmartPtr<const CD> p;                 // const object,
                                      // non-const pointer

const SmartPtr<CD> p = &goodCD;       // non-const object,
                                      // const pointer

const SmartPtr<const CD> p = &goodCD; // const object,
                                      // const pointer
```

Alas, this ointment has a fly in it. Using dumb pointers, we can initialize `const` pointers with non-`const` pointers and we can initialize pointers to `const` objects with pointers to non-`const`s; the rules for assignments are analogous. For example:

Click here to view code image

```
CD *pCD = new CD("Famous Movie Themes");
```

```
const CD * pConstCD = pCD;              // fine
```

But look what happens if we try the same thing with smart pointers:

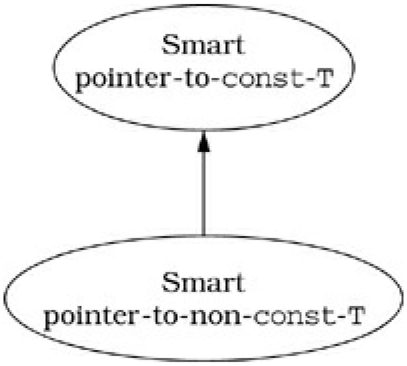
Click here to view code image

```
SmartPtr<CD> pCD = new CD("Famous Movie Themes");

SmartPtr<const CD> pConstCD = pCD;    // fine?
```

`SmartPtr<CD>` and `SmartPtr<const CD>` are completely different types. As far as your compilers know, they are unrelated, so they have no reason to believe they are assignment-compatible. In what must be an old story by now, the only way these two types will be considered assignment-compatible is if you've provided a function to convert objects of type `SmartPtr<CD>` to objects of type `SmartPtr<const CD>`. If you've got a compiler that supports member templates, you can use the technique shown above for automatically generating the implicit type conversion operators you need. (I remarked earlier that the technique worked anytime the corresponding conversion for dumb pointers would work, and I wasn't kidding. Conversions involving `const` are no exception.) If you don't have such a compiler, you have to jump through one more hoop.

Conversions involving `const` are a one-way street: it's safe to go from non-`const` to `const`, but it's not safe to go from `const` to non-`const`. Furthermore, anything you can do with a `const` pointer you can do with a non-`const` pointer, but with non-`const` pointers you can do other things, too (for example, assignment). Similarly, anything you can do with a pointer-to-`const` is legal for a pointer-to-non-`const`, but you can do some things (such as assignment) with pointers-to-non-`const`s that you can't do with pointers-to-`const`s.

These rules sound like the rules for public inheritance. You can convert from a derived class object to a base class object, but not vice versa, and you can do anything to a derived class object you can do to a base class object, but you can typically do additional things to a derived class object, as well. We can take advantage of this similarity when implementing smart pointers by having each smart pointer-to-`T` class publicly inherit from a corresponding smart pointer-to-`const`-`T` class:



[Click here to view code image](#)

```
template<class T>                // smart pointers to const
class SmartPtrToConst {          // objects

  ...                            // the usual smart pointer
                                 // member functions

protected:
  union {
    const T* constPointee;       // for SmartPtrToConst
access
    T* pointee;                  // for SmartPtr access
  };
};

template<class T>                // smart pointers to
class SmartPtr:                  // non-const objects
  public SmartPtrToConst<T> {
  ...                            // no data members
};
```

With this design, the smart pointer-to-non-`const`-`T` object needs to contain a dumb pointer-to-non-`const`-`T`, and the smart pointer-to-`const`-`T` needs to contain a dumb pointer-to-`const`-`T`. The naive way to handle this would be to put a dumb pointer-to-`const`-`T` in the base class and a dumb pointer-to-non-`const`-`T` in the derived class. That would be wasteful, however, because `SmartPtr` objects would contain two dumb pointers: the one they inherited from `SmartPtrToConst` and the one in `SmartPtr` itself.

This problem is resolved by employing that old battle axe of the C world, a union, which can be as useful in C++ as it is in C. The union is protected, so both classes have access to it, and it contains both of the necessary dumb pointer types. `SmartPtrToConst<T>` objects use the `constPointee` pointer, `SmartPtr<T>` objects use the `pointee` pointer. We therefore get the advantages of two different pointers without having to allocate space for more than one. Such is the beauty of a union. Of course, the member functions of the two classes must constrain themselves to using only the appropriate pointer, and you'll get no help from compilers in enforcing that constraint. Such is the risk of a union.

With this new design, we get the behavior we want:

Click here to view code image

```
SmartPtr<CD> pCD = new CD("Famous Movie Themes");

SmartPtrToConst<CD> pConstCD = pCD;   // fine
```

Evaluation

That wraps up the subject of smart pointers, but before we leave the topic, we should ask this question: are they worth the trouble, especially if your compilers lack support for member function templates?

Often they are. The reference-counting code of Item 29, for example, is greatly simplified by using smart pointers. Furthermore, as that example demonstrates, some uses of smart pointers are sufficiently limited in scope that things like testing for nullness, conversion to dumb pointers, inheritance-based conversions, and support for pointers-to-`const`s are irrelevant. At the same time, smart pointers can be tricky to implement, understand, and maintain. Debugging code using smart pointers is more difficult than debugging code using dumb pointers. Try as you may, you will never succeed in designing a general-purpose smart pointer that can seamlessly replace its dumb pointer counterpart.

Smart pointers nevertheless make it possible to achieve effects in your code that would otherwise be difficult to implement. Smart pointers should be used judiciously, but every C++ programmer will find them useful at one time or another.

Item 29: Reference counting

Reference counting is a technique that allows multiple objects with the same value to share a single representation of that value. There are two common motivations for the technique. The first is to simplify the bookkeeping surrounding heap objects. Once an object is allocated by calling `new`, it's crucial to keep track of who *owns* that object, because the owner — and only the owner — is responsible for calling `delete` on it. But ownership can be transferred from object to object as a program runs (by passing pointers as parameters, for example), so keeping track of an object's ownership is hard work. Classes like `auto_ptr` (see Item 9) can help with this task, but experience has shown that most programs still fail to get it right. Reference counting eliminates the burden of tracking object ownership, because when an object employs reference counting, it owns itself. When nobody is using it any longer, it destroys itself automatically. Thus, reference counting constitutes a simple form of garbage collection.

The second motivation for reference counting is simple common sense. If many objects have the same value, it's silly to store that value more than once. Instead, it's better to let all the objects with that value *share* its representation. Doing so not only saves memory, it also leads to faster-running programs, because there's no need to construct and destruct redundant copies of the same object value.

Like most simple ideas, this one hovers above a sea of interesting details. God may or may not be in the details, but successful implementations of reference counting certainly are. Before delving into details, however, let us master basics. A good way to begin is by seeing how we might come to have many objects with the same value in the first place. Here's one way:

Click here to view code image

```
class String {                // the standard string type may
public:                       // employ the techniques in this
                              // Item, but that is not required

  String(const char *value = "");
  String& operator=(const String& rhs);
  ...

private:
  char *data;
};

String a, b, c, d, e;
```

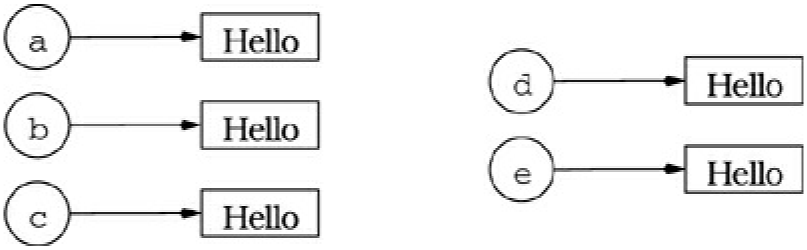
```
a = b = c = d = e = "Hello";
```

It should be apparent that objects `a` through `e` all have the same value, namely `"Hello"`. How that value is represented depends on how the `String` class is implemented, but a common implementation would have each `String` object carry its own copy of the value. For example, `String`'s assignment operator might be implemented like this:

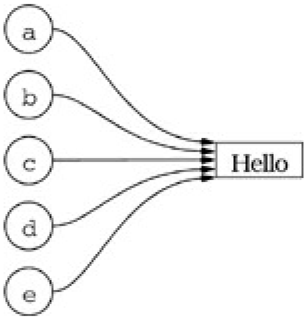
[Click here to view code image]

```
String& String::operator=(const String& rhs)
{
  if (this == &rhs) return *this;

  delete [] data;
  data = new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);

  return *this;
}
```

Given this implementation, we can envision the five objects and their values as follows:



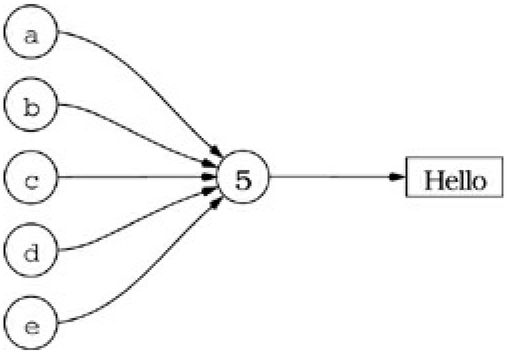
The redundancy in this approach is clear. In an ideal world, we'd like to change the picture to look like this:



Here only one copy of the value `"Hello"` is stored, and all the `String` objects with that value share its representation.

In practice, it isn't possible to achieve this ideal, because we need to keep track of how many objects are sharing a value. If object `a` above is assigned a different value from `"Hello"`, we can't destroy the value `"Hello"`, because four other objects still need it. On the other hand, if only a single object had the value `"Hello"` and that object went out of scope, no object would have that value and we'd have to destroy the value to avoid a resource leak.

The need to store information on the number of objects currently sharing — *referring to* — a value means our ideal picture must be modified somewhat to take into account the existence of a *reference count*:



(Some people call this number a *use count*, but I am not one of them. C++ has enough idiosyncrasies of its own; the last thing it needs is terminological factionalism.)

Implementing Reference Counting

Creating a reference-counted `String` class isn't difficult, but it does require attention to detail, so we'll walk through the implementation of the most common member functions of such a class. Before we do that, however, it's important to recognize that we need a place to store the reference count for each `String` value. That place cannot be in a `String` object, because we need one reference count per string *value*, not one reference count per string *object*. That implies a coupling between values and reference counts, so we'll create a class to store reference counts and the values they track. We'll call this class `StringValue`, and because its only *raison d'être* is to help implement the `String` class, we'll nest it inside `String`'s private section. Furthermore, it will be convenient to give all the member functions of `String` full access to the `StringValue` data structure, so we'll declare `StringValue` to be a `struct`. This is a trick worth knowing: nesting a struct in the private part of a class is a convenient way to give access to the struct to all the members of the class, but to deny access to everybody else (except, of course, friends of the class).

Our basic design looks like this:

Click here to view code image

```
class String {
public:
  ...                             // the usual String member
                                  // functions go here

private:
  struct StringValue { ... };     // holds a reference count
                                  // and a string value

  StringValue *value;             // value of this String
};
```

We could give this class a different name (`RCString`, perhaps) to emphasize that it's implemented using reference counting, but the implementation of a class shouldn't be of concern to clients of that class. Rather, clients should interest themselves only in a class's public interface. Our reference-counting implementation of the `String` interface supports exactly the same operations as a non-reference-counted version, so why muddy the conceptual waters by embedding implementation decisions in the names of classes that correspond to abstract concepts? Why indeed? So we don't.

Here's `StringValue`:

Click here to view code image

```
class String {
private:
  struct StringValue {
    size_t refCount;
    char *data;

    StringValue(const char *initValue);
    ~StringValue();

  };
```

```
    ...

  };

  String::StringValue::StringValue(const char *initValue)
  : refCount(1)
  {
    data = new char[strlen(initValue) + 1];
    strcpy(data, initValue);
  }

  String::StringValue::~StringValue()
  {
    delete [] data;
  }
```

That's all there is to it, and it should be clear that's nowhere near enough to implement the full functionality of a reference-counted string. For one thing, there's neither a copy constructor nor an assignment operator, and for another, there's no manipulation of the `refCount` field. Worry not — the missing functionality will be provided by the `String` class. The primary purpose of `StringValue` is to give us a place to associate a particular value with a count of the number of `String` objects sharing that value. `StringValue` gives us that, and that's enough.

We're now ready to walk our way through `String`'s member functions. We'll begin with the constructors:

Click here to view code image

```
  class String {
  public:
    String(const char *initValue = "");
    String(const String& rhs);

    ...

  };
```

The first constructor is implemented about as simply as possible. We use the passed-in `char*` string to create a new `StringValue` object, then we make the `String` object we're constructing point to the newly-minted `StringValue`:

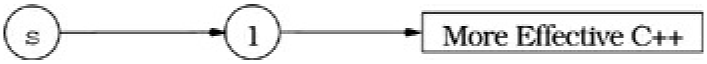
```
  String::String(const char *initValue)
```

```
: value(new StringValue(initValue))
{}
```

For client code that looks like this,

```
String s("More Effective C++");
```

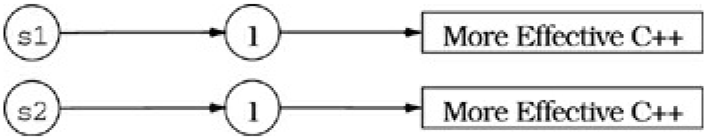
we end up with a data structure that looks like this:



`String` objects constructed separately, but with the same initial value do not share a data structure, so client code of this form,

```
String s1("More Effective C++");
String s2("More Effective C++");
```

yields this data structure:



It is possible to eliminate such duplication by having `String` (or `StringValue`) keep track of existing `StringValue` objects and create new ones only for truly unique strings, but such refinements on reference counting are somewhat off the beaten path. As a result, I'll leave them in the form of the feared and hated exercise for the reader.

The `String` copy constructor is not only unfeared and unhated, it's also efficient: the newly created `String` object shares the same `StringValue` object as the `String` object that's being copied:

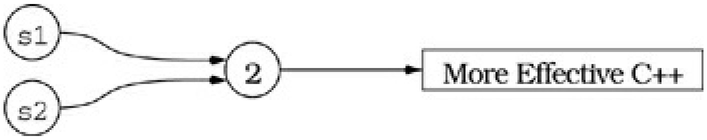
Click here to view code image

```
String::String(const String& rhs)
: value(rhs.value)
{
  ++value->refCount;
}
```

Graphically, code like this,

```
String s1("More Effective C++");
String s2 = s1;
```

results in this data structure:



This is substantially more efficient than a conventional (non-reference-counted) `String` class, because there is no need to allocate memory for the second copy of the string value, no need to deallocate that memory later, and no need to copy the value that would go in that memory. Instead, we merely copy a pointer and increment a reference count.

The `String` destructor is also easy to implement, because most of the time it doesn't do anything. As long as the reference count for a `StringValue` is non-zero, at least one `String` object is using the value; it must therefore not be destroyed. Only when the `String` being destructed is the sole user of the value — i.e., when the value's reference count is 1 — should the `String` destructor destroy the `StringValue` object:

Click here to view code image

```
class String {
public:
  ~String();
  ...
};

String::~String()
{
  if (--value->refCount == 0) delete value;
}
```

Compare the efficiency of this function with that of the destructor for a non-reference-counted implementation. Such a function would always call `delete` and would almost certainly have a nontrivial runtime cost. Provided that different `String` objects do in fact sometimes have the same values, the implementation above will sometimes do nothing more than decrement a counter and compare it to zero.

If, at this point, the appeal of reference counting is not becoming apparent, you're just not paying attention.

That's all there is to `String` construction and destruction, so we'll move on to consideration of the `String` assignment operator:

Click here to view code image

```
class String {
public:
  String& operator=(const String& rhs);
  ...

};
```

When a client writes code like this,

```
s1 = s2;                // s1 and s2 are both String
objects
```

the result of the assignment should be that `s1` and `s2` both point to the same `StringValue` object. That object's reference count should therefore be incremented during the assignment. Furthermore, the `StringValue` object that `s1` pointed to prior to the assignment should have its reference count decremented, because `s1` will no longer have that value. If `s1` was the only `String` with that value, the value should be destroyed. In C++, all that looks like this:

Click here to view code image

```
String& String::operator=(const String& rhs)
{
  if (value == rhs.value) {      // do nothing if the
values
    return *this;                // are already the same;
this
  }                              // subsumes the usual test
of
                                 // this against &rhs

  if (--value->refCount == 0) {  // destroy *this's value
if
    delete value;                // no one else is using it
  }

  value = rhs.value;             // have *this share rhs's
  ++value->refCount;             // value

  return *this;
}
```

Copy-on-Write

To round out our examination of reference-counted strings, consider an array-bracket operator (`[]`), which allows individual characters within strings to be read and written:

Click here to view code image

```
class String {
public:
  const char&
    operator[](int index) const;     // for const Strings

  char& operator[](int index);       // for non-const
Strings

  ...

};
```

Implementation of the `const` version of this function is straightforward, because it's a read-only operation; the value of the string can't be affected:

Click here to view code image

```
const char& String::operator[](int index) const
{
  return value->data[index];
}
```

(This function performs sanity checking on `index` in the grand C++ tradition, which is to say not at all. As usual, if you'd like a greater degree of parameter validation, it's easy to add.)

The non-`const` version of `operator[]` is a completely different story. This function may be called to read a character, but it might be called to write one, too:

Click here to view code image

```
String s;

...

cout << s[3];                    // this is a read
s[5] = 'x';                      // this is a write
```

We'd like to deal with reads and writes differently. A simple read can be dealt with in the same way as the `const` version of `operator[]` above, but a write must be implemented in quite a different fashion.

When we modify a `String`'s value, we have to be careful to avoid modifying the value of other `String` objects that happen to be sharing the same `StringValue` object. Unfortunately, there is no way for C++ compilers to tell us whether a particular use of `operator[]` is for a read or a write, so we must be pessimistic and assume that *all* calls to the non-`const operator[]` are for writes. (Proxy classes can help us differentiate reads from writes — see Item 30.)

To implement the non-`const operator[]` safely, we must ensure that no other `String` object shares the `StringValue` to be modified by the presumed write. In short, we must ensure that the reference count for a `String`'s `StringValue` object is exactly one any time we return a reference to a character inside that `StringValue` object. Here's how we do it:

Click here to view code image

```
char& String::operator[](int index)
{
  // if we're sharing a value with other String objects,
  // break off a separate copy of the value for ourselves
  if (value->refCount > 1) {
    --value->refCount;       // decrement current value's
                             // refCount, because we won't
                             // be using that value any more

    value =                                // make a copy of the
      new StringValue(value->data);        // value for ourselves
  }

  // return a reference to a character inside our
  // unshared StringValue object
  return value->data[index];
}
```

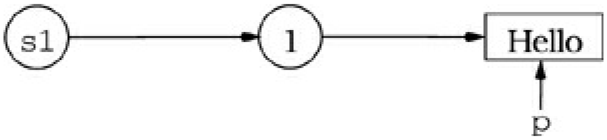
This idea — that of sharing a value with other objects until we have to write on our own copy of the value — has a long and distinguished history in Computer Science, especially in operating systems, where processes are routinely allowed to share pages until they want to modify data on their own copy of a page. The technique is common enough to have a name: *copy-on-write*. It's a specific example of a more general approach to efficiency, that of lazy evaluation (see Item 17).

Pointers, References, and Copy-on-Write

This implementation of copy-on-write allows us to preserve both efficiency and correctness — almost. There is one lingering problem. Consider this code:

```
String s1 = "Hello";

char *p = &s1[1];
```

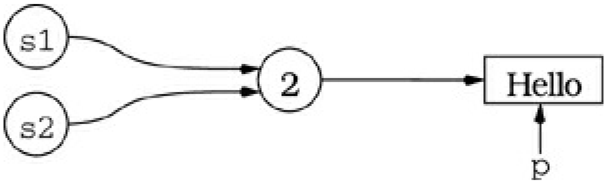
Our data structure at this point looks like this:



Now consider an additional statement:

```
String s2 = s1;
```

The `String` copy constructor will make `s2` share `s1`'s `StringValue`, so the resulting data structure will be this one:



The implications of a statement such as the following, then, are not pleasant to contemplate:

```
*p = 'x';                    // modifies both s1 and s2!
```

There is no way the `String` copy constructor can detect this problem, because it has no way to know that a pointer into `s1`'s `StringValue` object exists. And this problem isn't limited to pointers: it would exist if someone had saved a *reference* to the result of a call to `String`'s non-`const operator[]`.

There are at least three ways of dealing with this problem. The first is to ignore it, to pretend it doesn't exist. This approach turns out to be distressingly common in class libraries that implement reference-counted strings. If you have access to a reference-counted string, try the above example and see if you're distressed, too. If you're not sure if you have access to a reference-counted string, try the example anyway. Through the wonder of encapsulation, you may be using such a type without knowing it.

Not all implementations ignore such problems. A slightly more sophisticated way of dealing with such difficulties is to define them out of existence. Implementations adopting this strategy typically put something in their documentation that says, more or less, "Don't do that. If you do, results are undefined." If you then do it anyway — wittingly or no — and complain about the results, they respond, "Well, we *told* you not to do that." Such implementations are often efficient, but they leave much to be desired in the usability department.

There is a third solution, and that's to eliminate the problem. It's not difficult to implement, but it can reduce the amount of value sharing between objects. Its essence is this: add a flag to each `StringValue` object indicating whether that object is shareable. Turn the flag on initially (the object is shareable), but turn it off whenever the non-`const` `operator[]` is invoked on the value represented by that object. Once the flag is set to `false`, it stays that way forever.[†]

Here's a modified version of `StringValue` that includes a shareability flag:

Click here to view code image

```
class String {
private:
  struct StringValue {
    size_t refCount;
    bool shareable;                    // add this
    char *data;

    StringValue(const char *initValue);
    ~StringValue();

  };

...

};

String::StringValue::StringValue(const char *initValue)
:  refCount(1),
   shareable(true)                      // add this
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}

String::StringValue::~StringValue()
{
  delete [] data;
}
```

As you can see, not much needs to change; the two lines that require modification are flagged with comments. Of course, `String`'s member functions must be updated to take the `shareable` field into account. Here's how the copy constructor would do that:

Click here to view code image

```
String::String(const String& rhs)
{
  if (rhs.value->shareable) {
    value = rhs.value;
    ++value->refCount;
  }

  else {
    value = new StringValue(rhs.value->data);
  }
}
```

All the other `String` member functions would have to check the `shareable` field in an analogous fashion. The non-`const` version of `operator[]` would be the only function to set the `shareable` flag to `false`:

Click here to view code image

```
char& String::operator[](int index)
{
  if (value->refCount > 1) {
    --value->refCount;
    value = new StringValue(value->data);
  }

  value->shareable = false;     // add this

  return value->data[index];
}
```

If you use the proxy class technique of Item 30 to distinguish read usage from write usage in `operator[]`, you can usually reduce the number of `StringValue` objects that must be marked unshareable.

A Reference-Counting Base Class

Reference counting is useful for more than just strings. Any class in which different objects may have values in common is a legitimate candidate for reference counting. Rewriting a class to take advantage of reference counting can be a lot of work, however, and most of us already have more than enough to do. Wouldn't it be nice if we could somehow write (and test and document) the reference counting code in a context-independent manner, then just graft it onto classes when needed? Of course it would. In a curious twist of fate, there's a way to do it (or at least to do most of it).

The first step is to create a base class, `RCObject`, for reference-counted objects. Any class wishing to take advantage of automatic reference counting must inherit from this class. `RCObject` encapsulates the reference count itself, as well as functions for incrementing and decrementing that count. It also contains the code for destroying a value when it is no longer in use, i.e., when its reference count becomes 0. Finally, it contains a field that keeps track of whether this value is shareable, and it provides functions to query this value and set it to false. There is no need for a function to set the shareability field to true, because all values are shareable by default. As noted above, once an object has been tagged unshareable, there is no way to make it shareable again.

`RCObject`'s class definition looks like this:

Click here to view code image

```
class RCObject {
public:
  RCObject();
  RCObject(const RCObject& rhs);
  RCObject& operator=(const RCObject& rhs);
  virtual ~RCObject() = 0;
  void addReference();
  void removeReference();
```

```
    void markUnshareable();
    bool isShareable() const;

    bool isShared() const;

  private:
    size_t refCount;
    bool shareable;
  };
```

`RCObject`s can be created (as the base class parts of more derived objects) and destroyed; they can have new references added to them and can have current references removed; their shareability status can be queried and can be disabled; and they can report whether they are currently being shared. That's all they offer. As a class encapsulating the notion of being reference-countable, that's really all we have a right to expect them to do. Note the tell-tale virtual destructor, a sure sign this class is designed for use as a base class. Note also how the destructor is a *pure* virtual function, a sure sign this class is designed to be used *only* as a base class.

The code to implement `RCObject` is, if nothing else, brief:

Click here to view code image

```
  RCObject::RCObject()
  : refCount(0), shareable(true) {}

  RCObject::RCObject(const RCObject&)
  : refCount(0), shareable(true) {}

  RCObject& RCObject::operator=(const RCObject&)
  { return *this; }

  RCObject::~RCObject() {}        // virtual dtors must always
                                  // be implemented, even if
                                  // they are pure virtual
                                  // and do nothing (see also
                                  // Item 33)

  void RCObject::addReference() { ++refCount; }

  void RCObject::removeReference()
  { if (--refCount == 0) delete this; }

  void RCObject::markUnshareable()
```

```
{ shareable = false; }

bool RCObject::isShareable() const
{ return shareable; }

bool RCObject::isShared() const
{ return refCount > 1; }
```

Curiously, we set `refCount` to 0 inside both constructors. This seems counterintuitive. Surely at least the creator of the new `RCObject` is referring to it! As it turns out, it simplifies things for the creators of `RCObject`s to set `refCount` to 1 themselves, so we oblige them here by not getting in their way. We'll get a chance to see the resulting code simplification shortly.

Another curious thing is that the copy constructor always sets `refCount` to 0, regardless of the value of `refCount` for the `RCObject` we're copying. That's because we're creating a new object representing a value, and new values are always unshared and referenced only by their creator. Again, the creator is responsible for setting the `refCount` to its proper value.

The `RCObject` assignment operator looks downright subversive: it does *nothing*. Frankly, it's unlikely this operator will ever be called. `RCObject` is a base class for a shared *value* object, and in a system based on reference counting, such objects are not assigned to one another, objects *pointing* to them are. In our case, we don't expect `StringValue` objects to be assigned to one another, we expect only `String` objects to be involved in assignments. In such assignments, no change is made to the value of a `StringValue` — only the `StringValue` reference count is modified.

Nevertheless, it is conceivable that some as-yet-unwritten class might someday inherit from `RCObject` and might wish to allow assignment of reference-counted values (see Item 32). If so, `RCObject`'s assignment operator should do the right thing, and the right thing is to do nothing. To see why, imagine that we wished to allow assignments between `StringValue` objects. Given `StringValue` objects `sv1` and `sv2`, what should happen to `sv1`'s and `sv2`'s reference counts in an assignment?

Click here to view code image

```
sv1 = sv2;              // how are sv1's and sv2's reference
                        // counts affected?
```

Before the assignment, some number of `String` objects are pointing to `sv1`.

That number is unchanged by the assignment, because only `sv1`'s *value* changes. Similarly, some number of `String` objects are pointing to `sv2` prior to the assignment, and after the assignment, exactly the same `String` objects point to `sv2`. `sv2`'s reference count is also unchanged. When `RCObjects` are involved in an assignment, then, the number of objects pointing to those objects is unaffected, hence `RCObject::operator=` should change no reference counts. That's exactly what the implementation above does. Counterintuitive? Perhaps, but it's still correct.

The code for `RCObject::removeReference` is responsible not only for decrementing the object's `refCount`, but also for destroying the object if the new value of `refCount` is 0. It accomplishes this latter task by `delete`ing `this`, which, as Item 27 explains, is safe only if we know that `*this` is a heap object. For this class to be successful, we must engineer things so that `RCObject`s can be created only on the heap. General approaches to achieving that end are discussed in Item 27, but the specific measures we'll employ in this case are described at the conclusion of this Item.

To take advantage of our new reference-counting base class, we modify `StringValue` to inherit its reference counting capabilities from `RCObject`:

Click here to view code image

```
class String {
private:
  struct StringValue: public RCObject {
    char *data;

    StringValue(const char *initValue);
    ~StringValue();

  };

...

};

String::StringValue::StringValue(const char *initValue)
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}
```

```
String::StringValue::~StringValue()
{
  delete [] data;
}
```

This version of `StringValue` is almost identical to the one we saw earlier. The only thing that's changed is that `StringValue`'s member functions no longer manipulate the `refCount` field. `RCObject` now handles what they used to do.

Don't feel bad if you blanched at the sight of a nested class (`StringValue`) inheriting from a class (`RCObject`) that's unrelated to the nesting class (`String`). It looks weird to everybody at first, but it's perfectly kosher. A nested class is just as much a class as any other, so it has the freedom to inherit from whatever other classes it likes. In time, you won't think twice about such inheritance relationships.

Automating Reference Count Manipulations

The `RCObject` class gives us a place to store a reference count, and it gives us member functions through which that reference count can be manipulated, but the *calls* to those functions must still be manually inserted in other classes. It is still up to the `String` copy constructor and the `String` assignment operator to call `addReference` and `removeReference` on `StringValue` objects. This is clumsy. We'd like to move those calls out into a reusable class, too, thus freeing authors of classes like `String` from worrying about *any* of the details of reference counting. Can it be done? Isn't C++ supposed to support reuse?

It can, and it does. There's no easy way to arrange things so that *all* reference-counting considerations can be moved out of application classes, but there is a way to eliminate *most* of them for most classes. (In some application classes, you *can* eliminate all reference-counting code, but our `String` class, alas, isn't one of them. One member function spoils the party, and I suspect you won't be too surprised to hear it's our old nemesis, the non-`const` version of `operator[]`. Take heart, however; we'll tame that miscreant in the end.)

Notice that each `String` object contains a pointer to the `StringValue` object representing that `String`'s value:

Click here to view code image

```
class String {
private:
    struct StringValue: public RCObject { ... };

    StringValue *value;           // value of this String

    ...

  };
```

We have to manipulate the `refCount` field of the `StringValue` object anytime anything interesting happens to one of the pointers pointing to it. "Interesting happenings" include copying a pointer, reassigning one, and destroying one. If we could somehow make the *pointer itself* detect these happenings and automatically perform the necessary manipulations of the `refCount` field, we'd be home free. Unfortunately, pointers are rather dense creatures, and the chances of them detecting anything, much less automatically reacting to things they detect, are pretty slim. Fortunately, there's a way to smarten them up: replace them with objects that *act like* pointers, but that do more.

Such objects are called *smart pointers*, and you can read about them in more detail than you probably care to in Item 28. For our purposes here, it's enough to know that smart pointer objects support the member selection (`->`) and dereferencing (`*`) operations, just like real pointers (which, in this context, are generally referred to as *dumb pointers*), and, like dumb pointers, they are strongly typed: you can't make a smart pointer-to-T point to an object that isn't of type T.

Here's a template for objects that act as smart pointers to reference-counted objects:

Click here to view code image

```
  // template class for smart pointers-to-T objects. T must
  // support the RCObject interface, typically by inheriting
  // from RCObject
  template<class T>
  class RCPtr {
  public:
    RCPtr(T* realPtr = 0);
    RCPtr(const RCPtr& rhs);
    ~RCPtr();

    RCPtr& operator=(const RCPtr& rhs);
    T* operator->() const;        // see Item 28
    T& operator*() const;         // see Item 28

  private:
    T *pointee;                   // dumb pointer this
                                  // object is emulating

    void init();                  // common initialization
  };                              // code
```

This template gives smart pointer objects control over what happens during their construction, assignment, and destruction. When such events occur, these objects can automatically perform the appropriate manipulations of the `refCount` field in the objects to which they point.

For example, when an `RCPtr` is created, the object it points to needs to have its reference count increased. There's no need to burden application developers with the requirement to tend to this irksome detail manually, because `RCPtr` constructors can handle it themselves. The code in the two constructors is all but identical — only the member initialization lists differ — so rather than write it twice, we put it in a private member function called `init` and have both constructors call that:

Click here to view code image

```
  template<class T>
  RCPtr<T>::RCPtr(T* realPtr): pointee(realPtr)
  {
    init();
  }

  template<class T>
  RCPtr<T>::RCPtr(const RCPtr& rhs): pointee(rhs.pointee)
  {
    init();
  }

  template<class T>
  void RCPtr<T>::init()
  {
    if (pointee == 0) {              // if the dumb pointer is
      return;                        // null, so is the smart
  one
    }
  if (pointee->isShareable() == false) {  // if the value
    pointee = new T(*pointee);            // isn't
shareable,
  }                                       // copy it

  pointee->addReference();      // note that there is now a
}                               // new reference to the
value
```

Moving common code into a separate function like `init` is exemplary software engineering, but its luster dims when, as in this case, the function doesn't behave correctly.

The problem is this. When `init` needs to create a new copy of a value (because the existing copy isn't shareable), it executes the following code:

```
pointee = new T(*pointee);
```

The type of `pointee` is pointer-to-`T`, so this statement creates a new `T` object and initializes it by calling `T`'s copy constructor. In the case of an `RCPtr` in the `String` class, `T` will be `String::StringValue`, so the statement above will call `String::StringValue`'s copy constructor. We haven't declared a copy constructor for that class, however, so our compilers will generate one for us. The copy constructor so generated will, in accordance with the rules for automatically generated copy constructors in C++, copy only `StringValue`'s `data` *pointer*; it will *not* copy the `char*` string `data` points to. Such behavior is disastrous in nearly *any* class (not just reference-counted classes), and that's why you should get into the habit of writing a copy constructor (and an assignment operator) for all your classes that contain pointers.

The correct behavior of the `RCPtr<T>` template depends on `T` containing a copy constructor that makes a truly independent copy (i.e., a *deep copy*) of the value represented by `T`. We must augment `StringValue` with such a constructor before we can use it with the `RCPtr` class:

Click here to view code image

```
class String {
private:

  struct StringValue: public RCObject {
    StringValue(const StringValue& rhs);
    ...

  };

  ...

};

String::StringValue::StringValue(const StringValue& rhs)
{
  data = new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);
}
```

The existence of a deep-copying copy constructor is not the only assumption `RCPtr<T>` makes about `T`. It also requires that `T` inherit from `RCObject`, or at least that `T` provide all the functionality that `RCObject` does. In view of the fact that `RCPtr` objects are designed to point only to reference-counted objects, this is hardly an unreasonable assumption. Nevertheless, the assumption must be documented.

A final assumption in `RCPtr<T>` is that the type of the object pointed to is `T`. This seems obvious enough. After all, `pointee` is declared to be of type `T*`. But `pointee` might really point to a class *derived* from `T`. For example, if we had a class `SpecialStringValue` that inherited from `String::StringValue`,

Click here to view code image

```
class String {
private:
  struct StringValue: public RCObject { ... };

  struct SpecialStringValue: public StringValue { ... };

  ...

};
```

we could end up with a `String` containing a `RCPtr<StringValue>` pointing to a `SpecialStringValue` object. In that case, we'd want this part of `init`,

Click here to view code image

```
pointee = new T(*pointee);      // T is StringValue, but
                                // pointee really points to
                                // a SpecialStringValue
```

to call `SpecialStringValue`'s copy constructor, not `StringValue`'s. We can arrange for this to happen by using a virtual copy constructor (see Item 25). In the case of our `String` class, we don't expect classes to derive from `StringValue`, so we'll disregard this issue.

With `RCPtr`'s constructors out of the way, the rest of the class's functions can be dispatched with considerably greater alacrity. Assignment of an `RCPtr` is straightforward, though the need to test whether the newly assigned value is shareable complicates matters slightly. Fortunately, such complications have already been handled by the `init` function that was created for `RCPtr`'s constructors. We take advantage of that fact by using it again here:

Click here to view code image

```
template<class T>
RCPtr<T>& RCPtr<T>::operator=(const RCPtr& rhs)
{
  if (pointee != rhs.pointee) {     // skip assignments
                                    // where the value
                                    // doesn't change

    T *oldPointee = pointee;        // save old pointee
value

    pointee = rhs.pointee;          // point to new value
    init();                         // if possible, share
it
                                    // else make own copy

    if (oldPointee) {
      oldPointee->removeReference();// remove reference to
    }                               // current value
  return *this;
}
```

The destructor is easier. When an `RCPtr` is destroyed, it simply removes its reference to the reference-counted object:

Click here to view code image

```
template<class T>
RCPtr<T>::~RCPtr()
{
  if (pointee) pointee->removeReference();
}
```

If the `RCPtr` that just expired was the last reference to the object, that object will be destroyed inside `RCObject`'s `removeReference` member function. Hence `RCPtr` objects never need to worry about destroying the values they point to.

Finally, `RCPtr`'s pointer-emulating operators are part of the smart pointer boilerplate you can read about in Item 28:

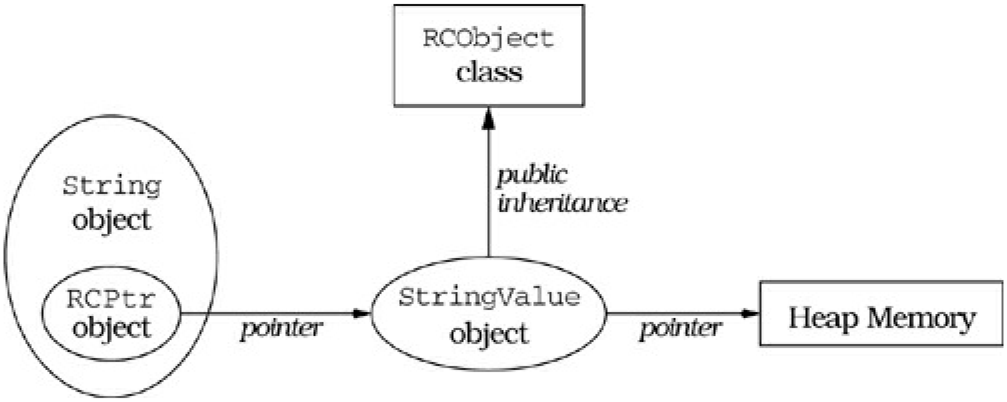
Click here to view code image

```
template<class T>
T* RCPtr<T>::operator->() const { return pointee; }

template<class T>
T& RCPtr<T>::operator*() const { return *pointee; }
```

Putting it All Together

Enough! *Finis*! At long last we are in a position to put all the pieces together and build a reference-counted `String` class based on the reusable `RCObject` and `RCPtr` classes. With luck, you haven't forgotten that that was our original goal.

Each reference-counted string is implemented via this data structure:



The classes making up this data structure are defined like this:

Click here to view code image

```
template<class T>                    // template class for smart
class RCPtr {                        // pointers-to-T objects; T
public:                              // must inherit from RCObject
  RCPtr(T* realPtr = 0);
  RCPtr(const RCPtr& rhs);
  ~RCPtr();
  RCPtr& operator=(const RCPtr& rhs);

  T* operator->() const;
  T& operator*() const;

private:
  T *pointee;

  void init();
};
class RCObject {                // base class for reference-
public:                         // counted objects
  RCObject();
  RCObject(const RCObject& rhs);
  RCObject& operator=(const RCObject& rhs);
  virtual ~RCObject() = 0;

  void addReference();
  void removeReference();

  void markUnshareable();
  bool isShareable() const;

  bool isShared() const;

private:
  size_t refCount;
  bool shareable;
};

class String {                  // class to be used by
public:                         // application developers

  String(const char *value = "");

  const char& operator[](int index) const;
  char& operator[](int index);

private:
  // class representing string values
  struct StringValue: public RCObject {
    char *data;

    StringValue(const char *initValue);
    StringValue(const StringValue& rhs);
    void init(const char *initValue);
    ~StringValue();
  };

  RCPtr<StringValue> value;
};
```

For the most part, this is just a recap of what we've already developed, so nothing should be much of a surprise. Close examination reveals we've added an `init` function to `String::StringValue`, but, as we'll see below, that serves the same purpose as the corresponding function in `RCPtr`: it prevents code duplication in the constructors.

There is a significant difference between the public interface of this `String` class and the one we used at the beginning of this Item. Where is the copy constructor? Where is the assignment operator? Where is the destructor? Something is definitely amiss here.

Actually, no. Nothing is amiss. In fact, some things are working perfectly. If you don't see what they are, prepare yourself for a C++ epiphany.

We don't need those functions anymore. Sure, copying of `String` objects is still supported, and yes, the copying will correctly handle the underlying reference-counted `StringValue` objects, but the `String` class doesn't have to provide a single line of code to make this happen. That's because the compiler-generated copy constructor for `String` will automatically call the copy constructor for `String`'s `RCPtr` member, and the copy constructor for *that* class will perform all the necessary manipulations of the `StringValue` object, including its reference count. An `RCPtr` is a *smart* pointer, remember? We designed it to take care of the details of reference counting, so that's what it does. It also handles assignment and destruction, and that's why `String` doesn't need to write those functions, either. Our original goal was to move the unreusable reference-counting code out of our hand-written `String` class and into context-independent classes where it would be available for use with *any* class. Now we've done it (in the form of the `RCObject` and `RCPtr` classes), so don't be so surprised when it suddenly starts working. It's *supposed* to work.

Just so you have everything in one place, here's the implementation of `RCObject`:

Click here to view code image

```
RCObject::RCObject()
: refCount(0), shareable(true) {}

RCObject::RCObject(const RCObject&)
: refCount(0), shareable(true) {}

RCObject& RCObject::operator=(const RCObject&)
{ return *this; }

RCObject::~RCObject() {}

void RCObject::addReference() { ++refCount; }

void RCObject::removeReference()
{ if (--refCount == 0) delete this; }

void RCObject::markUnshareable()
{ shareable = false; }

bool RCObject::isShareable() const
{ return shareable; }

bool RCObject::isShared() const
{ return refCount > 1; }
```

And here's the implementation of `RCPtr`:

Click here to view code image

```
template<class T>
void RCPtr<T>::init()
{
  if (pointee == 0) return;

  if (pointee->isShareable() == false) {
    pointee = new T(*pointee);
  }

  pointee->addReference();
}

template<class T>
RCPtr<T>::RCPtr(T* realPtr)
: pointee(realPtr)
{ init(); }
```

```
template<class T>
RCPtr<T>::RCPtr(const RCPtr& rhs)
: pointee(rhs.pointee)
{ init(); }

template<class T>
RCPtr<T>::~RCPtr()
{ if (pointee) pointee->removeReference(); }

template<class T>
RCPtr<T>& RCPtr<T>::operator=(const RCPtr& rhs)
{
  if (pointee != rhs.pointee) {
    T *oldPointee = pointee;
    pointee = rhs.pointee;
    init();
    if (oldPointee) oldPointee->removeReference();
  }

  return *this;
}

template<class T>
T* RCPtr<T>::operator->() const { return pointee; }

template<class T>
T& RCPtr<T>::operator*() const { return *pointee; }
```

The implementation of `String::StringValue` looks like this:

Click here to view code image

```
void String::StringValue::init(const char *initValue)
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}

String::StringValue::StringValue(const char *initValue)
{ init(initValue); }

String::StringValue::StringValue(const StringValue& rhs)
{ init(rhs.data); }

String::StringValue::~StringValue()
{ delete [] data; }
```

Ultimately, all roads lead to `String`, and that class is implemented this way:

Click here to view code image

```
String::String(const char *initValue)
: value(new StringValue(initValue)) {}

const char& String::operator[](int index) const
{ return value->data[index]; }

char& String::operator[](int index)
{
  if (value->isShared()) {
    value = new StringValue(value->data);
  }

  value->markUnshareable();

  return value->data[index];
}
```

If you compare the code for this `String` class with that we developed for the `String` class using dumb pointers, you'll be struck by two things. First, there's a lot less of it here than there. That's because `RCPtr` has assumed much of the reference-counting burden that used to fall on `String`. Second, the code that remains in `String` is nearly unchanged: the smart pointer replaced the dumb pointer essentially seamlessly. In fact, the only changes are in `operator[]`, where we call `isShared` instead of checking the value of `refCount` directly and where our use of the smart `RCPtr` object eliminates the need to manually manipulate the reference count during a copy-on-write.

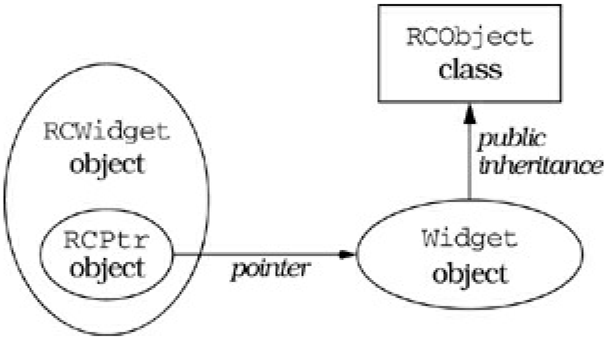
This is all very nice, of course. Who can object to less code? Who can oppose encapsulation success stories? The bottom line, however, is determined more by the impact of this newfangled `String` class on its clients than by any of its implementation details, and it is here that things really shine. If no news is good news, the news here is very good indeed. *The `String` interface has not changed*. We added reference counting, we added the ability to mark individual string values as unshareable, we moved the notion of reference countability into a new base class, we added smart pointers to automate the manipulation of reference counts, yet not one line of client code needs to be changed. Sure, we changed the `String` class definition, so clients who want to take advantage of reference-counted strings must recompile and relink, but their investment in code is completely and utterly preserved. You see? Encapsulation really *is* a wonderful thing.

Adding Reference Counting to Existing Classes

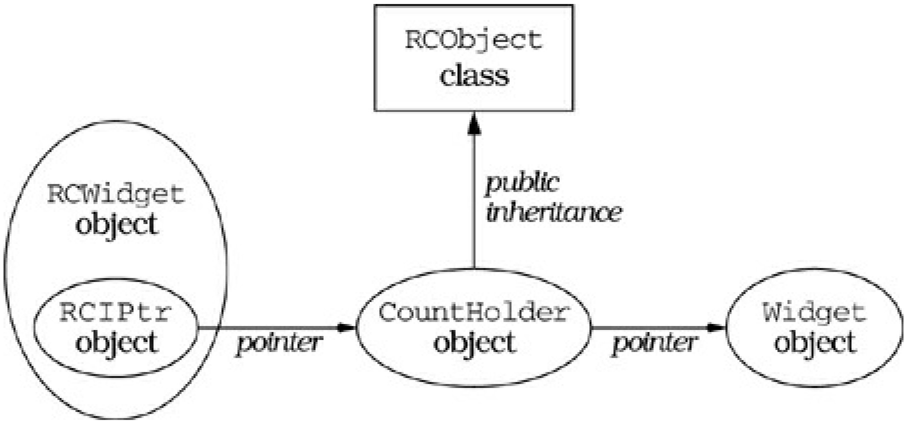
Everything we've discussed so far assumes we have access to the source code of the classes we're interested in. But what if we'd like to apply the benefits of reference counting to some class `Widget` that's in a library we can't modify? There's no way to make `Widget` inherit from `RCObject`, so we can't use smart `RCPtr`s with it. Are we out of luck?

We're not. With some minor modifications to our design, we can add reference counting to *any* type.

First, let's consider what our design would look like if we could have `Widget` inherit from `RCObject`. In that case, we'd have to add a class, `RCWidget`, for clients to use, but everything would then be analogous to our `String`/`StringValue` example, with `RCWidget` playing the role of `String` and `Widget` playing the role of `StringValue`. The design would look like this:



We can now apply the maxim that most problems in Computer Science can be solved with an additional level of indirection. We add a new class, `CountHolder`, to hold the reference count, and we have `CountHolder` inherit from `RCObject`. We also have `CountHolder` contain a pointer to a `Widget`. We then replace the smart `RCPtr` template with an equally smart `RCIPtr` template that knows about the existence of the `CountHolder` class. (The "I" in `RCIPtr` stands for "indirect.") The modified design looks like this:


RCObject class
public inheritance
RCWidget object
RCIPtr object
pointer
CountHolder object
pointer
Widget object

Just as `StringValue` was an implementation detail hidden from clients of `String`, `CountHolder` is an implementation detail hidden from clients of `RCWidget`. In fact, it's an implementation detail of `RCIPtr`, so it's nested inside that class. `RCIPtr` is implemented this way:

Click here to view code image

```
template<class T>
class RCIPtr {
public:
  RCIPtr(T* realPtr = 0);
  RCIPtr(const RCIPtr& rhs);
  ~RCIPtr();

  RCIPtr& operator=(const RCIPtr& rhs);

  T* operator->() const;
  T& operator*() const;

  RCObject& getRCObject()       // give clients access to
  { return *counter; }          // isShared, etc.

private:
  struct CountHolder: public RCObject {
    ~CountHolder() { delete pointee; }
    T *pointee;
  };

  CountHolder *counter;

  void init();
};
```

```
template<class T>
void RCIPtr<T>::init()
{
  if (counter->isShareable() == false) {
    T *oldValue = counter->pointee;
    counter = new CountHolder;
    counter->pointee = oldValue ? new T(*oldValue) : 0;
  }

  counter->addReference();
}

template<class T>
RCIPtr<T>::RCIPtr(T* realPtr)
: counter(new CountHolder)
{
  counter->pointee = realPtr;
  init();
}

template<class T>
RCIPtr<T>::RCIPtr(const RCIPtr& rhs)
: counter(rhs.counter)
{ init(); }

template<class T>
RCIPtr<T>::~RCIPtr()
{ counter->removeReference(); }

template<class T>
RCIPtr<T>& RCIPtr<T>::operator=(const RCIPtr& rhs)
{
  if (counter != rhs.counter) {
    counter->removeReference();
    counter = rhs.counter;
    init();
  }
  return *this;
}

template<class T>
T* RCIPtr<T>::operator->() const
{ return counter->pointee; }

template<class T>
```

```
T& RCIPtr<T>::operator*() const
{ return *(counter->pointee); }
```

If you compare this implementation with that of `RCPtr`, you'll see they are conceptually identical. They differ only in that `RCPtr` objects point to values directly, while `RCIPtr` objects point to values through an intervening `CountHolder` object.

Given `RCIPtr`, it's easy to implement `RCWidget`, because each function in `RCWidget` is implemented by forwarding the call through the underlying `RCIPtr` to a `Widget` object. For example, if `Widget` looks like this,

[Click here to view code image]

```
class Widget {
public:
  Widget(int size);
  Widget(const Widget& rhs);
  ~Widget();

  Widget& operator=(const Widget& rhs);

  void doThis();
  int showThat() const;
};
```

`RCWidget` will be defined this way:

[Click here to view code image]

```
class RCWidget {
public:
  RCWidget(int size): value(new Widget(size)) {}

  void doThis()
  {
    if (value.getRCObject().isShared()) { // do COW if
      value = new Widget(*value);         // Widget is
shared
    }
    value->doThis();
  }
  int showThat() const { return value->showThat(); }
private:
  RCIPtr<Widget> value;
};
```

Note how the `RCWidget` constructor calls the `Widget` constructor (via the `new` operator — see Item 8) with the argument it was passed; how `RCWidget`'s `doThis` calls `doThis` in the `Widget` class; and how `RCWidget::showThat` returns whatever its `Widget` counterpart returns. Notice also how `RCWidget` declares no copy constructor, no assignment operator, and no destructor. As with the `String` class, there is no need to write these functions. Thanks to the behavior of the `RCIPtr` class, the default versions do the right things.

If the thought occurs to you that creation of `RCWidget` is so mechanical, it could be automated, you're right. It would not be difficult to write a program that takes a class like `Widget` as input and produces a class like `RCWidget` as output. If you write such a program, please let me know.

Evaluation

Let us disentangle ourselves from the details of widgets, strings, values, smart pointers, and reference-counting base classes. That gives us an opportunity to step back and view reference counting in a broader context. In that more general context, we must address a higher-level question, namely, when is reference counting an appropriate technique?

Reference-counting implementations are not without cost. Each reference-counted value carries a reference count with it, and most operations require that this reference count be examined or manipulated in some way. Object values therefore require more memory, and we sometimes execute more code when we work with them. Furthermore, the underlying source code is considerably more complex for a reference-counted class than for a less elaborate implementation. An un-reference-counted string class typically stands on its own, while our final `String` class is useless unless it's augmented with three auxiliary classes (`StringValue`, `RCObject`, and `RCPtr`). True, our more complicated design holds out the promise of greater efficiency when values can be shared, it eliminates the need to track object ownership, and it promotes reusability of the reference counting idea and implementation. Nevertheless, that quartet of classes has to be written, tested, documented, and maintained, and that's going to be more work than writing, testing, documenting, and maintaining a single class. Even a manager can see that.

Reference counting is an optimization technique predicated on the assumption that objects will commonly share values (see also Item 18). If this assumption fails to hold, reference counting will use more memory than a more conventional implementation and it will execute more code. On the other hand, if your objects *do* tend to have common values, reference counting should save you both time and space. The bigger your object values and the more objects that can simultaneously share values, the more memory you'll save. The more you copy and assign values between objects, the more time you'll save. The more expensive it is to create and destroy a value, the more time you'll save there, too. In short, reference counting is most useful for improving efficiency under the following conditions:

- **Relatively few values are shared by relatively many objects.** Such sharing typically arises through calls to assignment operators and copy constructors. The higher the objects/values ratio, the better the case for reference counting.
- **Object values are expensive to create or destroy, or they use lots of memory.** Even when this is the case, reference counting still buys you nothing unless these values can be shared by multiple objects.

There is only one sure way to tell whether these conditions are satisfied, and that way is *not* to guess or rely on your programmer's intuition (see Item 16). The reliable way to find out whether your program can benefit from reference counting is to profile or instrument it. That way you can find out if creating and destroying values is a performance bottleneck, and you can measure the objects/values ratio. Only when you have such data in hand are you in a position to determine whether the benefits of reference counting (of which there are many) outweigh the disadvantages (of which there are also many).

Even when the conditions above are satisfied, a design employing reference counting may still be inappropriate. Some data structures (e.g., directed graphs) lead to self-referential or circular dependency structures. Such data structures have a tendency to spawn isolated collections of objects, used by no one, whose reference counts never drop to zero. That's because each object in the unused structure is pointed to by at least one other object in the same structure. Industrial-strength garbage collectors use special techniques to find such structures and eliminate them, but the simple reference-counting approach we've examined here is not easily extended to include such techniques.

Reference counting can be attractive even if efficiency is not your primary concern. If you find yourself weighed down with uncertainty over who's allowed to delete what, reference counting could be just the technique you need to ease your burden. Many programmers are devoted to reference counting for this reason alone.

Let us close this discussion on a technical note by tying up one remaining loose end. When `RCObject::removeReference` decrements an object's reference count, it checks to see if the new count is 0. If it is, `removeReference` destroys the object by `delete`ing `this`. This is a safe operation only if the object was allocated by calling `new`, so we need some way of ensuring that `RCObject`s are created only in that manner.

In this case we do it by convention. `RCObject` is designed for use as a base class of reference-counted value objects, and those value objects should be referred to only by smart `RCPtr` pointers. Furthermore, the value objects should be instantiated only by application objects that realize values are being shared; the classes describing the value objects should never be available for general use. In our example, the class for value objects is `StringValue`, and we limit its use by making it private in `String`. Only `String` can create `StringValue` objects, so it is up to the author of the `String` class to ensure that all such objects are allocated via `new`.

Our approach to the constraint that `RCObject`s be created only on the heap, then, is to assign responsibility for conformance to this constraint to a well-defined set of classes and to ensure that only that set of classes can create `RCObject`s. There is no possibility that random clients can accidently (or maliciously) create `RCObject`s in an inappropriate manner. We limit the right to create reference-counted objects, and when we do hand out the right, we make it clear that it's accompanied by the concomitant responsibility to follow the rules governing object creation.

Item 30: Proxy classes

Though your in-laws may be one-dimensional, the world, in general, is not. Unfortunately, C++ hasn't yet caught on to that fact. At least, there's little evidence for it in the language's support for arrays. You can create two-dimensional, three-dimensional — heck, you can create *n*-dimensional — arrays in FORTRAN, in BASIC, even in COBOL (okay, FORTRAN only allows up to seven dimensions, but let's not quibble), but can you do it in C++? Only sometimes, and even then only sort of.

This much is legal:

```
int data[10][20];                   // 2D array: 10 by 20
```

The corresponding construct using variables as dimension sizes, however, is not:

Click here to view code image

```
void processInput(int dim1, int dim2)
{
  int data[dim1][dim2];          // error! array dimensions
  ...                            // must be known during
}                                // compilation
```

It's not even legal for a heap-based allocation:

Click here to view code image

```
int *data =
  new int[dim1][dim2];           // error!
```

Implementing Two-Dimensional Arrays

Multidimensional arrays are as useful in C++ as they are in any other language, so it's important to come up with a way to get decent support for them. The usual way is the standard one in C++: create a class to represent the objects we need but that are missing in the language proper. Hence we can define a class template for two-dimensional arrays:

```
template<class T>
class Array2D {
public:
    Array2D(int dim1, int dim2);
    ...

  };
```

Now we can define the arrays we want:

Click here to view code image

```
  Array2D<int> data(10, 20);            // fine

  Array2D<float> *data =
    new Array2D<float>(10, 20);         // fine

  void processInput(int dim1, int dim2)
  {
    Array2D<int> data(dim1, dim2);      // fine
    ...
  }
```

Using these array objects, however, isn't quite as straightforward. In keeping with the grand syntactic tradition of both C and C++, we'd like to be able to use brackets to index into our arrays,

```
  cout << data[3][6];
```

but how do we declare the indexing operator in `Array2D` to let us do this?

Our first impulse might be to declare `operator[][]` functions, like this:

Click here to view code image

```
  template<class T>
  class Array2D {
  public:

    // declarations that won't compile
    T& operator[][](int index1, int index2);
    const T& operator[][](int index1, int index2) const;

    ...

  };
```

We'd quickly learn to rein in such impulses, however, because there is no such thing as `operator[][]`, and don't think your compilers will forget it. (For a complete list of operators, overloadable and otherwise, see Item 7.) We'll have to do something else.

If you can stomach the syntax, you might follow the lead of the many programming languages that use parentheses to index into arrays. To use parentheses, you just overload `operator()`:

Click here to view code image

```
template<class T>
class Array2D {
public:

  // declarations that will compile
  T& operator()(int index1, int index2);
  const T& operator()(int index1, int index2) const;

  ...

};
```

Clients then use arrays this way:

```
cout << data(3, 6);
```

This is easy to implement and easy to generalize to as many dimensions as you like. The drawback is that your `Array2D` objects don't look like built-in arrays any more. In fact, the above access to element (3, 6) of `data` looks, on the face of it, like a function call.

If you reject the thought of your arrays looking like FORTRAN refugees, you might turn again to the notion of using brackets as the indexing operator. Although there is no such thing as `operator[][]`, it is nonetheless legal to write code that appears to use it:

```
int data[10][20];

...

cout << data[3][6];                // fine
```

What gives?

What gives is that the variable `data` is not really a two-dimensional array at all, it's a 10-element one-dimensional array. Each of those 10 elements is itself a 20-element array, so the expression `data[3][6]` really means `(data[3])[6]`, i.e., the seventh element of the array that is the fourth element of `data`. In short, the value yielded by the first application of the brackets is another array, so the second application of the brackets gets an element from that secondary array.

We can play the same game with our `Array2D` class by overloading `operator[]` to return an object of a new class, `Array1D`. We can then overload `operator[]` again in `Array1D` to return an element in our original two-dimensional array:

[Click here to view code image]

```
template<class T>
class Array2D {
public:
  class Array1D {
  public:
    T& operator[](int index);
    const T& operator[](int index) const;
    ...

  };

  Array1D operator[](int index);
  const Array1D operator[](int index) const;
  ...

};
```

The following then becomes legal:

[Click here to view code image]

```
Array2D<float> data(10, 20);

...

cout << data[3][6];              // fine
```

Here, `data[3]` yields an `Array1D` object and the `operator[]` invocation on that object yields the float in position (3, 6) of the original two-dimensional array.

Clients of the `Array2D` class need not be aware of the presence of the `Array1D` class. Objects of this latter class stand for one-dimensional array objects that, conceptually, do not exist for clients of `Array2D`. Such clients program as if they were using real, live, honest-to-Allah two-dimensional arrays. It is of no concern to `Array2D` clients that those objects must, in order to satisfy the vagaries of C++, be syntactically compatible with one-dimensional arrays of other one-dimensional arrays.

Each `Array1D` object *stands for* a one-dimensional array that is absent from the conceptual model used by clients of `Array2D`. Objects that stand for other objects are often called *proxy objects*, and the classes that give rise to proxy objects are often called *proxy classes*. In this example, `Array1D` is a proxy class. Its instances stand for one-dimensional arrays that, conceptually, do not exist. (The terminology for proxy objects and classes is far from universal; objects of such classes are also sometimes known as *surrogates*.)

Distinguishing Reads from Writes via `operator[]`

The use of proxies to implement classes whose instances act like multidimensional arrays is common, but proxy classes are more flexible than that. Item 5, for example, shows how proxy classes can be employed to prevent single-argument constructors from being used to perform unwanted type conversions. Of the varied uses of proxy classes, however, the most heralded is that of helping distinguish reads from writes through `operator[]`.

Consider a reference-counted string type that supports `operator[]`. Such a type is examined in detail in Item 29. If the concepts behind reference counting have slipped your mind, it would be a good idea to familiarize yourself with the material in that Item now.

A string type supporting `operator[]` allows clients to write code like this:

Click here to view code image

```
String s1, s2;              // a string-like class; the
                            // use of proxies keeps this
                            // class from conforming to
                            // the standard string
...                         // interface

cout << s1[5];              // read s1

s2[5] = 'x';                // write s2

s1[3] = s2[8];              // write s1, read s2
```

Note that `operator[]` can be called in two different contexts: to read a character or to write a character. Reads are known as *rvalue* usages; writes are known as *lvalue* usages. (The terms come from the field of compilers, where an lvalue goes on the left-hand side of an assignment and an rvalue goes on the right-hand side.) In general, using an object as an lvalue means using it such that it might be modified, and using it as an rvalue means using it such that it cannot be modified.

We'd like to distinguish between lvalue and rvalue usage of `operator[]` because, especially for reference-counted data structures, reads can be much less expensive to implement than writes. As Item 29 explains, writes of reference-counted objects may involve copying an entire data structure, but reads never require more than the simple returning of a value. Unfortunately, inside `operator[]`, there is no way to determine the context in which the function was called; it is not possible to distinguish lvalue usage from rvalue usage within `operator[]`.

"But wait," you say, "we don't need to. We can overload `operator[]` on the basis of its `const`ness, and that will allow us to distinguish reads from writes." In other words, you suggest we solve our problem this way:

Click here to view code image

```
class String {
public:
  const char& operator[](int index) const;      // for reads
  char& operator[](int index);                  // for
writes
  ...

};
```

Alas, this won't work. Compilers choose between `const` and non-`const` member functions by looking only at whether the *object* invoking a function is `const`. No consideration is given to the context in which a call is made. Hence:

Click here to view code image

```
String s1, s2;

...

cout << s1[5];                     // calls non-const
operator[],
                                   // because s1 isn't const
```

```
s2[5] = 'x';                  // also calls non-const
                              // operator[]: s2 isn't const

s1[3] = s2[8];                // both calls are to non-
const
                              // operator[], because both
s1
                              // and s2 are non-const
objects
```

Overloading `operator[]`, then, fails to distinguish reads from writes.

In Item 29, we resigned ourselves to this unsatisfactory state of affairs and made the conservative assumption that all calls to `operator[]` were for writes. This time we shall not give up so easily. It may be impossible to distinguish lvalue from rvalue usage inside `operator[]`, but we still want to do it. We will therefore find a way. What fun is life if you allow yourself to be limited by the possible?

Our approach is based on the fact that though it may be impossible to tell whether `operator[]` is being invoked in an lvalue or an rvalue context from within `operator[]`, we can still treat reads differently from writes if we *delay* our lvalue-versus-rvalue actions until we see how the result of `operator[]` is used. All we need is a way to postpone our decision on whether our object is being read or written until *after* `operator[]` has returned. (This is an example of *lazy evaluation* — see Item 17.)

A proxy class allows us to buy the time we need, because we can modify `operator[]` to return a *proxy* for a string character instead of a string character itself. We can then wait to see how the proxy is used. If it's read, we can belatedly treat the call to `operator[]` as a read. If it's written, we must treat the call to `operator[]` as a write.

We will see the code for this in a moment, but first it is important to understand the proxies we'll be using. There are only three things you can do with a proxy:

- Create it, i.e., specify which string character it stands for.
- Use it as the target of an assignment, in which case you are really making an assignment to the string character it stands for. When used in this way, a proxy represents an lvalue use of the string on which

`operator[]` was invoked.

- Use it in any other way. When used like this, a proxy represents an rvalue use of the string on which `operator[]` was invoked.

Here are the class definitions for a reference-counted `String` class using a proxy class to distinguish between lvalue and rvalue usages of `operator[]`:

Click here to view code image

```
class String {                  // reference-counted strings;
public:                         // see Item 29 for details

  class CharProxy {             // proxies for string chars
  public:
    CharProxy(String& str, int index);          // creation

    CharProxy& operator=(const CharProxy& rhs); // lvalue
    CharProxy& operator=(char c);               // uses

    operator char() const;                      // rvalue
                                                // use
  private:
    String& theString;          // string this proxy pertains to

    int charIndex;              // char within that string
                                // this proxy stands for
  };

  // continuation of String class
  const CharProxy
    operator[](int index) const;     // for const Strings
  CharProxy operator[](int index);   // for non-const
Strings
  ...

friend class CharProxy;

private:
  RCPtr<StringValue> value;
};
```

Other than the addition of the `CharProxy` class (which we'll examine below), the only difference between this `String` class and the final `String` class in Item 29 is that both `operator[]` functions now return `CharProxy` objects. Clients of

`String` can generally ignore this, however, and program as if the `operator[]` functions returned characters (or references to characters — see Item 1) in the usual manner:

Click here to view code image

```
String s1, s2;                // reference-counted strings
                              // using proxies
...

cout << s1[5];                // still legal, still works

s2[5] = 'x';                  // also legal, also works

s1[3] = s2[8];                // of course it's legal,
                              // of course it works
```

What's interesting is not that this works. What's interesting is *how* it works.

Consider first this statement:

```
cout << s1[5];
```

The expression `s1[5]` yields a `CharProxy` object. No output operator is defined for such objects, so your compilers labor to find an implicit type conversion they can apply to make the call to `operator<<` succeed (see Item 5). They find one: the implicit conversion from `CharProxy` to `char` declared in the `CharProxy` class. They automatically invoke this conversion operator, and the result is that the string character represented by the `CharProxy` is printed. This is representative of the `CharProxy`-to-`char` conversion that takes place for all `CharProxy` objects used as rvalues.

Lvalue usage is handled differently. Look again at

```
s2[5] = 'x';
```

As before, the expression `s2[5]` yields a `CharProxy` object, but this time that object is the target of an assignment. Which assignment operator is invoked? The target of the assignment is a `CharProxy`, so the assignment operator that's called is in the `CharProxy` class. This is crucial, because inside a `CharProxy` assignment operator, we know that the `CharProxy` object being assigned to is being used as an lvalue. We therefore know that the string character for which the proxy stands is being used as an lvalue, and we must take whatever actions are necessary to implement lvalue access for that character.

Similarly, the statement

```
s1[3] = s2[8];
```

calls the assignment operator for two `CharProxy` objects, and inside that operator we know the object on the left is being used as an lvalue and the object on the right as an rvalue.

"Yeah, yeah, yeah," you grumble, "show me." Okay. Here's the code for `String`'s `operator[]` functions:

Click here to view code image

```
const String::CharProxy String::operator[](int index) const
{
  return CharProxy(const_cast<String&>(*this), index);
}
String::CharProxy String::operator[](int index)
{
  return CharProxy(*this, index);
}
```

Each function just creates and returns a proxy for the requested character. No action is taken on the character itself: we defer such action until we know whether the access is for a read or a write.

Note that the `const` version of `operator[]` returns a `const` proxy. Because `CharProxy::operator=` isn't a `const` member function, such proxies can't be used as the target of assignments. Hence neither the proxy returned from the `const` version of `operator[]` nor the character for which it stands may be used as an lvalue. Conveniently enough, that's exactly the behavior we want for the `const` version of `operator[]`.

Note also the use of a `const_cast` (see Item 2) on `*this` when creating the `CharProxy` object that the `const operator[]` returns. That's necessary to satisfy the constraints of the `CharProxy` constructor, which accepts only a non-`const` `String`. Casts are usually worrisome, but in this case the `CharProxy` object returned by `operator[]` is itself `const`, so there is no risk the `String` containing the character to which the proxy refers will be modified.

Each proxy returned by an `operator[]` function remembers which string it pertains to and, within that string, the index of the character it represents:

Click here to view code image

```
String::CharProxy::CharProxy(String& str, int index)
: theString(str), charIndex(index) {}
```

Conversion of a proxy to an rvalue is straightforward — we just return a copy of the character represented by the proxy:

Click here to view code image

```
String::CharProxy::operator char() const
{
  return theString.value->data[charIndex];
}
```

If you've forgotten the relationship among a `String` object, its `value` member, and the `data` member it points to, you can refresh your memory by turning to Item 29. Because this function returns a character by value, and because C++ limits the use of such by-value returns to rvalue contexts only, this conversion function can be used only in places where an rvalue is legal.

We thus turn to implementation of `CharProxy`'s assignment operators, which is where we must deal with the fact that a character represented by a proxy is being used as the target of an assignment, i.e., as an lvalue. We can implement `CharProxy`'s conventional assignment operator as follows:

Click here to view code image

```
String::CharProxy&
String::CharProxy::operator=(const CharProxy& rhs)
{
  // if the string is sharing a value with other String
objects,
  // break off a separate copy of the value for this string
only
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value-
>data);
  }

  // now make the assignment: assign the value of the char
  // represented by rhs to the char represented by *this
  theString.value->data[charIndex] =
    rhs.theString.value->data[rhs.charIndex];

  return *this;
}
```

If you compare this with the implementation of the non-`const` `String`::`operator[]` in Item 29 on page 207, you'll see that they are strikingly similar. This is to be expected. In Item 29, we pessimistically assumed that all invocations of the non-`const operator[]` were writes, so we treated them as such. Here, we moved the code implementing a write into `CharProxy`'s assignment operators, and that allows us to avoid paying for a write when the non-`const operator[]` is used only in an rvalue context. Note, by the way, that this function requires access to `String`'s private data member `value`. That's why `CharProxy` is declared a friend in the earlier class definition for `String`.

The second `CharProxy` assignment operator is almost identical:

Click here to view code image

```
String::CharProxy& String::CharProxy::operator=(char c)
{
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value-
>data);
  }

  theString.value->data[charIndex] = c;

  return *this;
}
```

As an accomplished software engineer, you would, of course, banish the code duplication present in these two assignment operators to a private `CharProxy` member function that both would call. Aren't you the modular one?

Limitations

The use of a proxy class is a nice way to distinguish lvalue and rvalue usage of `operator[]`, but the technique is not without its drawbacks. We'd like proxy objects to seamlessly replace the objects they stand for, but this ideal is difficult to achieve. That's because objects are used as lvalues in contexts other than just assignment, and using proxies in such contexts usually yields behavior different from using real objects.

Consider again the code fragment from Item 29 that motivated our decision to add a shareability flag to each `StringValue` object. If `String`::`operator[]` returns a `CharProxy` instead of a `char&`, that code will no longer compile:

```
String s1 = "Hello";

char *p = &s1[1];            // error!
```

The expression `s1[1]` returns a `CharProxy`, so the type of the expression on the right-hand side of the "`=`" is `CharProxy*`. There is no conversion from a `CharProxy*` to a `char*`, so the initialization of `p` fails to compile. In general, taking the address of a proxy yields a different type of pointer than does taking the address of a real object.

To eliminate this difficulty, you'll need to overload the address-of operators for the `CharProxy` class:

Click here to view code image

```
class String {
public:

  class CharProxy {
  public:
    ...
    char * operator&();
    const char * operator&() const;
    ...
  };

  ...
};
```

These functions are easy to implement. The `const` function just returns a pointer to a `const` version of the character represented by the proxy:

Click here to view code image

```
const char * String::CharProxy::operator&() const
{
  return &(theString.value->data[charIndex]);
}
```

The non-`const` function is a bit more work, because it returns a pointer to a character that may be modified. This is analogous to the behavior of the non-`const` version of `String`::`operator[]` in Item 29, and the implementation is equally analogous:

Click here to view code image

```
char * String::CharProxy::operator&()
```

```
{
  // make sure the character to which this function returns
  // a pointer isn't shared by any other String objects
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value-
>data);
  }

  // we don't know how long the pointer this function
  // returns will be kept by clients, so the StringValue
  // object can never be shared
  theString.value->markUnshareable();

  return &(theString.value->data[charIndex]);
}
```

Much of this code is common to other `CharProxy` member functions, so I know you'd encapsulate it in a private member function that all would call.

A second difference between `char`s and the `CharProxy`s that stand for them becomes apparent if we have a template for reference-counted arrays that use proxy classes to distinguish lvalue and rvalue invocations of `operator[]`:

Click here to view code image

```
template<class T>                 // reference-counted array
class Array {                     // using proxies
public:
  class Proxy {
  public:
    Proxy(Array<T>& array, int index);
    Proxy& operator=(const T& rhs);
    operator T() const;
    ...
  };

  const Proxy operator[](int index) const;
  Proxy operator[](int index);
  ...
};
```

Consider how these arrays might be used:

Click here to view code image

```
Array<int> intArray;
```

```
...
intArray[5] = 22;                    // fine
intArray[5] += 5;                    // error!
++intArray[5];                       // error!
```

As expected, use of `operator[]` as the target of a simple assignment succeeds, but use of `operator[]` on the left-hand side of a call to `operator+=` or `operator++` fails. That's because `operator[]` returns a proxy, and there is no `operator+=` or `operator++` for `Proxy` objects. A similar situation exists for other operators that require lvalues, including `operator*=`, `operator<<=`, `operator--`, etc. If you want these operators to work with `operator[]` functions that return proxies, you must define each of these functions for the `Array<T>::Proxy` class. That's a lot of work, and you probably don't want to do it. Unfortunately, you either do the work or you do without. Them's the breaks.

A related problem has to do with invoking member functions on real objects through proxies. To be blunt about it, you can't. For example, suppose we'd like to work with reference-counted arrays of rational numbers. We could define a class `Rational` and then use the `Array` template we just saw:

Click here to view code image

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  int numerator() const;
  int denominator() const;
  ...
};

Array<Rational> array;
```

This is how we'd expect to be able to use such arrays, but, alas, we'd be disappointed:

Click here to view code image

```
cout << array[4].numerator();             // error!
int denom = array[22].denominator();      // error!
```

By now the difficulty is predictable; `operator[]` returns a proxy for a rational number, not an actual `Rational` object. But the `numerator` and `denominator` member functions exist only for `Rational`s, not their proxies. Hence the complaints by your compilers. To make proxies behave like the objects they stand for, you must overload each function applicable to the real objects so it applies to proxies, too.

Yet another situation in which proxies fail to replace real objects is when being passed to functions that take references to non-`const` objects:

Click here to view code image

```
void swap(char& a, char& b);    // swaps the value of a and
b

String s = "+C+";               // oops, should be "C++"

swap(s[0], s[1]);               // this should fix the
                                // problem, but it won't
                                // compile
```

`String`::`operator[]` returns a `CharProxy`, but `swap` demands that its arguments be of type `char&`. A `CharProxy` may be implicitly converted into a `char`, but there is no conversion function to a `char&`. Furthermore, the `char` to which it may be converted can't be bound to `swap`'s `char&` parameters, because that `char` is a temporary object (it's `operator char`'s return value) and, as Item 19 explains, there are good reasons for refusing to bind temporary objects to non-`const` reference parameters.

A final way in which proxies fail to seamlessly replace real objects has to do with implicit type conversions. When a proxy object is implicitly converted into the real object it stands for, a user-defined conversion function is invoked. For instance, a `CharProxy` can be converted into the `char` it stands for by calling `operator char`. As Item 5 explains, compilers may use only one user-defined conversion function when converting a parameter at a call site into the type needed by the corresponding function parameter. As a result, it is possible for function calls that succeed when passed real objects to fail when passed proxies. For example, suppose we have a `TVStation` class and a function, `watchTV`:

Click here to view code image

```
class TVStation {
public:
  TVStation(int channel);
```

```
    ...
  };
  void watchTV(const TVStation& station, float hoursToWatch);
```

Thanks to implicit type conversion from `int` to `TVStation` (see Item 5), we could then do this:

Click here to view code image

```
  watchTV(10, 2.5);              // watch channel 10 for
                                 // 2.5 hours
```

Using the template for reference-counted arrays that use proxy classes to distinguish lvalue and rvalue invocations of `operator[]`, however, we could not do this:

Click here to view code image

```
  Array<int> intArray;
  intArray[4] = 10;
  watchTV(intArray[4], 2.5);     // error! no conversion
                                 // from Proxy<int> to
                                 // TVStation
```

Given the problems that accompany implicit type conversions, it's hard to get too choked up about this. In fact, a better design for the `TVStation` class would declare its constructor `explicit`, in which case even the first call to `watchTV` would fail to compile. For all the details on implicit type conversions and how `explicit` affects them, see Item 5.

Evaluation

Proxy classes allow you to achieve some types of behavior that are otherwise difficult or impossible to implement. Multidimensional arrays are one example, lvalue/rvalue differentiation is a second, suppression of implicit conversions (see Item 5) is a third.

At the same time, proxy classes have disadvantages. As function return values, proxy objects are temporaries (see Item 19), so they must be created and destroyed. That's not free, though the cost may be more than recouped through their ability to distinguish write operations from read operations. The very existence of proxy classes increases the complexity of software systems that employ them, because additional classes make things harder to design, implement, understand, and maintain, not easier.

Finally, shifting from a class that works with real objects to a class that works with proxies often changes the semantics of the class, because proxy objects usually exhibit behavior that is subtly different from that of the real objects they represent. Sometimes this makes proxies a poor choice when designing a system, but in many cases there is little need for the operations that would make the presence of proxies apparent to clients. For instance, few clients will want to take the address of an `Array1D` object in the two-dimensional array example we saw at the beginning of this Item, and there isn't much chance that an `ArraySize` object (see Item 5) would be passed to a function expecting a different type. In many cases, proxies can stand in for real objects perfectly acceptably. When they can, it is often the case that nothing else will do.

Item 31: Making functions virtual with respect to more than one object

Sometimes, to borrow a phrase from Jacqueline Susann, once is not enough. Suppose, for example, you're bucking for one of those high-profile, high-prestige, high-paying programming jobs at that famous software company in Redmond, Washington — by which of course I mean Nintendo. To bring yourself to the attention of Nintendo's management, you might decide to write a video game. Such a game might take place in outer space and involve space ships, space stations, and asteroids.

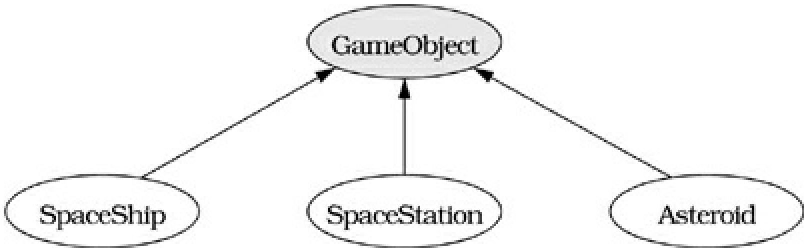
As the ships, stations, and asteroids whiz around in your artificial world, they naturally run the risk of colliding with one another. Let's assume the rules for such collisions are as follows:

- If a ship and a station collide at low velocity, the ship docks at the station. Otherwise the ship and the station sustain damage that's proportional to the speed at which they collide.
- If a ship and a ship or a station and a station collide, both participants in the collision sustain damage that's proportional to the speed at which they hit.

- If a small asteroid collides with a ship or a station, the asteroid is destroyed. If it's a big asteroid, the ship or the station is destroyed.
- If an asteroid collides with another asteroid, both break into pieces and scatter little baby asteroids in all directions.

This may sound like a dull game, but it suffices for our purpose here, which is to consider how to structure the C++ code that handles collisions between objects.

We begin by noting that ships, stations, and asteroids share some common features. If nothing else, they're all in motion, so they all have a velocity that describes that motion. Given this commonality, it is natural to define a base class from which they all inherit. In practice, such a class is almost invariably an abstract base class, and, if you heed the warning I give in Item 33, base classes are always abstract. The hierarchy might therefore look like this:



Click here to view code image

```
class GameObject { ... };

class SpaceShip: public GameObject { ... };

class SpaceStation: public GameObject { ... };

class Asteroid: public GameObject { ... };
```

Now, suppose you're deep in the bowels of your program, writing the code to check for and handle object collisions. You might come up with a function that looks something like this:

Click here to view code image

```
void checkForCollision(GameObject& object1,
                       GameObject& object2)
{
  if (theyJustCollided(object1, object2)) {
      processCollision(object1, object2);
    }
    else {
      ...
    }
  }
```

This is where the programming challenge becomes apparent. When you call `processCollision`, you know that `object1` and `object2` just collided, and you know that what happens in that collision depends on what `object1` really is and what `object2` really is, but you don't know what kinds of objects they really are; all you know is that they're both `GameObject`s. If the collision processing depended only on the dynamic type of `object1`, you could make `processCollision` virtual in `GameObject` and call `object1.processCollision(object2)`. You could do the same thing with `object2` if the details of the collision depended only on its dynamic type. What happens in the collision, however, depends on *both* their dynamic types. A function call that's virtual on only one object, you see, is not enough.

What you need is a kind of function whose behavior is somehow virtual on the types of more than one object. C++ offers no such function. Nevertheless, you still have to implement the behavior required above. The question, then, is how you are going to do it.

One possibility is to scrap the use of C++ and choose another programming language. You could turn to CLOS, for example, the Common Lisp Object System. CLOS supports what is possibly the most general object-oriented function-invocation mechanism one can imagine: *multi-methods*. A multi-method is a function that's virtual on as many parameters as you'd like, and CLOS goes even further by giving you substantial control over how calls to overloaded multi-methods are resolved.

Let us assume, however, that you must implement your game in C++ — that you must come up with your own way of implementing what is commonly referred to as *double-dispatching*. (The name comes from the object-oriented programming community, where what C++ programmers know as a virtual function call is termed a "message dispatch." A call that's virtual on two parameters is implemented through a "double dispatch." The generalization of this — a function acting virtual on several parameters — is called *multiple dispatch*.) There are several approaches you might consider. None is without its disadvantages, but that shouldn't surprise you. C++ offers no direct support for double-dispatching, so you must yourself do the work compilers do when they implement virtual functions (see Item 24). If that were easy to do, we'd probably all be doing it ourselves and simply programming in C. We aren't and we don't, so fasten your seat belts, it's going to be a bumpy ride.

Using Virtual Functions and RTTI

Virtual functions implement a single dispatch; that's half of what we need; and compilers do virtual functions for us, so we begin by declaring a virtual function `collide` in `GameObject`. This function is overridden in the derived classes in the usual manner:

Click here to view code image

```
class GameObject {
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};

class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);
  ...
};
```

Here I'm showing only the derived class `SpaceShip`, but `SpaceStation` and `Asteroid` are handled in exactly the same manner.

The most common approach to double-dispatching returns us to the unforgiving world of virtual function emulation via chains of `if-then-else`s. In this harsh world, we first discover the real type of `otherObject`, then we test it against all the possibilities:

Click here to view code image

```
// if we collide with an object of unknown type, we
// throw an exception of this type:
class CollisionWithUnknownObject {
public:
  CollisionWithUnknownObject(GameObject& whatWeHit);
  ...
};
```

```
void SpaceShip::collide(GameObject& otherObject)
{
  const type_info& objectType = typeid(otherObject);

  if (objectType == typeid(SpaceShip)) {
    SpaceShip& ss = static_cast<SpaceShip&>(otherObject);

    process a SpaceShip-SpaceShip collision;
  }

  else if (objectType == typeid(SpaceStation)) {
    SpaceStation& ss =
      static_cast<SpaceStation&>(otherObject);

    process a SpaceShip-SpaceStation collision;
  }

  else if (objectType == typeid(Asteroid)) {
    Asteroid& a = static_cast<Asteroid&>(otherObject);

    process a SpaceShip-Asteroid collision;

  }

  else {
    throw CollisionWithUnknownObject(otherObject);
  }
}
```

Notice how we need to determine the type of only one of the objects involved in the collision. The other object is `*this`, and its type is determined by the virtual function mechanism. We're inside a `SpaceShip` member function, so `*this` must be a `SpaceShip` object. Thus we only have to figure out the real type of `otherObject`.

There's nothing complicated about this code. It's easy to write. It's even easy to make work. That's one of the reasons RTTI is worrisome: it looks harmless. The true danger in this code is hinted at only by the final `else` clause and the exception that's thrown there.

We've pretty much bidden *adios* to encapsulation, because each `collide` function must be aware of each of its sibling classes, i.e., those classes that inherit from `GameObject`. In particular, if a new type of object — a new class —

is added to the game, we must update each RTTI-based `if-then-else` chain in the program that might encounter the new object type. If we forget even a single one, the program will have a bug, and the bug will *not* be obvious. Furthermore, compilers are in no position to help us detect such an oversight, because they have no idea what we're doing.

This kind of type-based programming has a long history in C, and one of the things we know about it is that it yields programs that are essentially unmaintainable. Enhancement of such programs eventually becomes unthinkable. This is the primary reason why virtual functions were invented in the first place: to shift the burden of generating and maintaining type-based function calls from programmers to compilers. When we employ RTTI to implement double-dispatching, we are harking back to the bad old days.

The techniques of the bad old days led to errors in C, and they'll lead to errors in C++, too. In recognition of our human frailty, we've included a final `else` clause in the `collide` function, a clause where control winds up if we hit an object we don't know about. Such a situation is, in principle, impossible, but where were our principles when we decided to use RTTI? There are various ways to handle such unanticipated interactions, but none is very satisfying. In this case, we've chosen to throw an exception, but it's not clear how our callers can hope to handle the error any better than we can, since we've just run into something we didn't know existed.

Using Virtual Functions Only

There is a way to minimize the risks inherent in an RTTI approach to implementing double-dispatching, but before we look at that, it's convenient to see how to attack the problem using nothing but virtual functions. That strategy begins with the same basic structure as the RTTI approach. The `collide` function is declared virtual in `GameObject` and is redefined in each derived class. In addition, `collide` is overloaded in each class, one overloading for each derived class in the hierarchy:

Click here to view code image

```
class SpaceShip;              // forward declarations
class SpaceStation;
class Asteroid;
class GameObject {
public:
  virtual void collide(GameObject&   otherObject) = 0;
  virtual void collide(SpaceShip&    otherObject) = 0;
  virtual void collide(SpaceStation& otherObject) = 0;
  virtual void collide(Asteroid&     otherobject) = 0;
  ...

};

class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject&   otherObject);
  virtual void collide(SpaceShip&    otherObject);
  virtual void collide(SpaceStation& otherObject);
  virtual void collide(Asteroid&     otherobject);
  ...

};
```

The basic idea is to implement double-dispatching as two single dispatches, i.e., as two separate virtual function calls: the first determines the dynamic type of the first object, the second determines that of the second object. As before, the first virtual call is to the `collide` function taking a `GameObject&` parameter. That function's implementation now becomes startlingly simple:

Click here to view code image

```
void SpaceShip::collide(GameObject& otherObject)
{
  otherObject.collide(*this);
}
```

At first glance, this appears to be nothing more than a recursive call to `collide` with the order of the parameters reversed, i.e., with `otherObject` becoming the object calling the member function and `*this` becoming the function's parameter. Glance again, however, because this is *not* a recursive call. As you know, compilers figure out which of a set of functions to call on the basis of the static types of the arguments passed to the function. In this case, four different `collide` functions could be called, but the one chosen is based on the static type of `*this`. What is that static type? Being inside a member function of the class `SpaceShip`, `*this` must be of type `SpaceShip`. The call is therefore to the `collide` function taking a `SpaceShip&`, not the `collide` function taking a `GameObject&`.

All the `collide` functions are virtual, so the call inside `SpaceShip::collide` resolves to the implementation of `collide` corresponding to the real type of `otherObject`. Inside *that* implementation of `collide`, the real types of both objects are known, because the left-hand object is `*this` (and therefore has as its type the class implementing the member function) and the right-hand object's real type is `SpaceShip`, the same as the declared type of the parameter.

All this may be clearer when you see the implementations of the other `collide` functions in `SpaceShip`:

Click here to view code image

```
void SpaceShip::collide(SpaceShip& otherObject)
{
  process a SpaceShip-SpaceShip collision;
}

void SpaceShip::collide(SpaceStation& otherObject)
{
  process a SpaceShip-SpaceStation collision;
}

void SpaceShip::collide(Asteroid& otherObject)
{
  process a SpaceShip-Asteroid collision;
}
```

As you can see, there's no muss, no fuss, no RTTI, no need to throw exceptions for unexpected object types. There can be no unexpected object types — that's the whole point of using virtual functions. In fact, were it not for its fatal flaw, this would be the perfect solution to the double-dispatching problem.

The flaw is one it shares with the RTTI approach we saw earlier: each class must know about its siblings. As new classes are added, the code must be updated. However, the *way* in which the code must be updated is different in this case. True, there are no `if-then-else`s to modify, but there is something that is often worse: each class definition must be amended to include a new virtual function. If, for example, you decide to add a new class `Satellite` (inheriting from `GameObject`) to your game, you'd have to add a new `collide` function to each of the existing classes in the program.

Modifying existing classes is something you are frequently in no position to do. If, instead of writing the entire video game yourself, you started with an off-the-shelf class library comprising a video game application framework, you might not have write access to the `GameObject` class or the framework classes derived from it. In that case, adding new member functions, virtual or otherwise, is not an option. Alternatively, you may have *physical* access to the classes requiring modification, but you may not have *practical* access. For example, suppose you *were* hired by Nintendo and were put to work on programs using a library containing `GameObject` and other useful classes. Surely you wouldn't be the only one using that library, and Nintendo would probably be less than thrilled about recompiling every application using that library each time you decided to add a new type of object to your program. In practice, libraries in wide use are modified only rarely, because the cost of recompiling everything using those libraries is too great.

The long and short of it is if you need to implement double-dispatching in your program, your best recourse is to modify your design to eliminate the need. Failing that, the virtual function approach is safer than the RTTI strategy, but it constrains the extensibility of your system to match that of your ability to edit header files. The RTTI approach, on the other hand, makes no recompilation demands, but, if implemented as shown above, it generally leads to software that is unmaintainable. You pays your money and you takes your chances.

Emulating Virtual Function Tables

There is a way to improve those chances. You may recall from Item 24 that compilers typically implement virtual functions by creating an array of function pointers (the vtbl) and then indexing into that array when a virtual function is called. Using a vtbl eliminates the need for compilers to perform chains of `if-then-else`-like computations, and it allows compilers to generate the same code at all virtual function call sites: determine the correct vtbl index, then call the function pointed to at that position in the vtbl.

There is no reason you can't do this yourself. If you do, you not only make your RTTI-based code more efficient (indexing into an array and following a function pointer is almost always more efficient than running through a series of `if-then-else` tests, and it generates less code, too), you also isolate the use of RTTI to a single location: the place where your array of function pointers is initialized. I should mention that the meek may inherit the earth, but the meek of heart may wish to take a few deep breaths before reading what follows.

We begin by making some modifications to the functions in the `GameObject` hierarchy:

Click here to view code image

```
class GameObject {
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};

class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);
  virtual void hitSpaceShip(SpaceShip& otherObject);
  virtual void hitSpaceStation(SpaceStation& otherObject);
  virtual void hitAsteroid(Asteroid& otherobject);
  ...
};
void SpaceShip::hitSpaceShip(SpaceShip& otherObject)
{
  process a SpaceShip-SpaceShip collision;
}

void SpaceShip::hitSpaceStation(SpaceStation& otherObject)
{
  process a SpaceShip-SpaceStation collision;
}

void SpaceShip::hitAsteroid(Asteroid& otherObject)
{
  process a SpaceShip-Asteroid collision;
}
```

Like the RTTI-based hierarchy we started out with, the `GameObject` class contains only one function for processing collisions, the one that performs the first of the two necessary dispatches. Like the virtual-function-based hierarchy we saw later, each kind of interaction is encapsulated in a separate function, though in this case the functions have different names instead of sharing the name `collide`. There is a reason for this abandonment of overloading, and we shall see it soon. For the time being, note that the design above contains everything we need except an implementation for

`SpaceShip::collide`; that's where the various `hit` functions will be invoked. As before, once we successfully implement the `SpaceShip` class, the `SpaceStation` and `Asteroid` classes will follow suit.

Inside `SpaceShip::collide`, we need a way to map the dynamic type of the parameter `otherObject` to a member function pointer that points to the appropriate collision-handling function. An easy way to do this is to create an associative array that, given a class name, yields the appropriate member function pointer. It's possible to implement `collide` using such an associative array directly, but it's a bit easier to understand what's going on if we add an intervening function, `lookup`, that takes a `GameObject` and returns the appropriate member function pointer. That is, you pass `lookup` a `GameObject`, and it returns a pointer to the member function to call when you collide with something of that `GameObject`'s type.

Here's the declaration of `lookup`:

Click here to view code image

```
class SpaceShip: public GameObject {
private:
  typedef void (SpaceShip::*HitFunctionPtr)(GameObject&);

  static HitFunctionPtr lookup(const GameObject&
whatWeHit);

  ...
};
```

The syntax of function pointers is never very pretty, and for member function pointers it's worse than usual, so we've `typedef`ed `HitFunctionPtr` to be shorthand for a pointer to a member function of `SpaceShip` that takes a `GameObject&` and returns nothing.

Once we've got `lookup`, implementation of `collide` becomes the proverbial piece of cake:

Click here to view code image

```
void SpaceShip::collide(GameObject& otherObject)
{
  HitFunctionPtr hfp =
    lookup(otherObject);         // find the function to
call
```

```
    if (hfp) {                        // if a function was found
      (this->*hfp)(otherObject);   // call it
    }
    else {
      throw CollisionWithUnknownObject(otherObject);
    }
  }
```

Provided we've kept the contents of our associative array in sync with the class hierarchy under `GameObject`, `lookup` must always find a valid function pointer for the object we pass it. People are people, however, and mistakes have been known to creep into even the most carefully crafted software systems. That's why we still check to make sure a valid pointer was returned from `lookup`, and that's why we still throw an exception if the impossible occurs and the lookup fails.

All that remains now is the implementation of `lookup`. Given an associative array that maps from object types to member function pointers, the lookup itself is easy, but creating, initializing, and destroying the associative array is an interesting problem of its own.

Such an array should be created and initialized before it's used, and it should be destroyed when it's no longer needed. We could use `new` and `delete` to create and destroy the array manually, but that would be error-prone: how could we guarantee the array wasn't used before we got around to initializing it? A better solution is to have compilers automate the process, and we can do that by making the associative array static in `lookup`. That way it will be created and initialized the first time `lookup` is called, and it will be automatically destroyed sometime after `main` is exited.

Furthermore, we can use the `map` template from the Standard Template Library (see Item 35) as the associative array, because that's what a `map` is:

Click here to view code image

```
  class SpaceShip: public GameObject {
  private:
    typedef void (SpaceShip::*HitFunctionPtr)(GameObject&);
    typedef map<string, HitFunctionPtr> HitMap;
    ...
  };
```

```
SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap;
  ...
}
```

Here, `collisionMap` is our associative array. It maps the name of a class (as a `string` object) to a `SpaceShip` member function pointer. Because `map<string, HitFunctionPtr>` is quite a mouthful, we use a typedef to make it easier to swallow. (For fun, try writing the declaration of `collisionMap` without using the `HitMap` and `HitFunctionPtr` typedefs. Most people will want to do this only once.)

Given `collisionMap`, the implementation of `lookup` is rather anticlimactic. That's because searching for something is an operation directly supported by the `map` class, and the one member function we can always (portably) call on the result of a `typeid` invocation is `name` (which, predictably[†], yields the name of the object's dynamic type). To implement `lookup`, then, we just find the entry in `collisionMap` corresponding to the dynamic type of `lookup`'s argument.

The code for `lookup` is straightforward, but if you're not familiar with the Standard Template Library (again, see Item 35), it may not seem that way. Don't worry. The comments in the function explain what's going on.

Click here to view code image

```
SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap;  // we'll see how to
                               // initialize this below

  // look up the collision-processing function for the type
  // of whatWeHit. The value returned is a pointer-like
  // object called an "iterator" (see Item 35).
  HitMap::iterator mapEntry=
    collisionMap.find(typeid(whatWeHit).name());

  // mapEntry == collisionMap.end() if the lookup failed;
  // this is standard map behavior. Again, see Item 35.
  if (mapEntry == collisionMap.end()) return 0;

  // If we get here, the search succeeded. mapEntry
```

```
    // points to a complete map entry, which is a
    // (string, HitFunctionPtr) pair. We want only the
    // second part of the pair, so that's what we return.
    return (*mapEntry).second;
  }
```

The final statement in the function returns `(*mapEntry).second` instead of the more conventional `mapEntry->second` in order to satisfy the vagaries of the STL. For details, see page 96B.

Initializing Emulated Virtual Function Tables

Which brings us to the initialization of `collisionMap`. We'd like to say something like this,

Click here to view code image

```
  // An incorrect implementation
  SpaceShip::HitFunctionPtr
  SpaceShip::lookup(const GameObject& whatWeHit)
  {
    static HitMap collisionMap;

    collisionMap["SpaceShip"] = &hitSpaceShip;
    collisionMap["SpaceStation"] = &hitSpaceStation;
    collisionMap["Asteroid"] = &hitAsteroid;

    ...
  }
```

but this inserts the member function pointers into collisionMap *each time* `lookup` is called, and that's needlessly inefficient. In addition, this won't compile, but that's a secondary problem we'll address shortly.

What we need now is a way to put the member function pointers into `collisionMap` only once — when `collisionMap` is created. That's easy enough to accomplish; we just write a private static member function called `initializeCollisionMap` to create and initialize our `map`, then we initialize `collisionMap` with `initializeCollisionMap`'s return value:

Click here to view code image

```
  class SpaceShip: public GameObject {
  private:
    static HitMap initializeCollisionMap();
  ...

};

SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap = initializeCollisionMap();
  ...
}
```

But this means we may have to pay the cost of copying the `map` object returned from `initializeCollisionMap` into `collisionMap` (see Items 19 and 20). We'd prefer not to do that. We wouldn't have to pay if `initializeCollisionMap` returned a pointer, but then we'd have to worry about making sure the `map` object the pointer pointed to was destroyed at an appropriate time.

Fortunately, there's a way for us to have it all. We can turn `collisionMap` into a smart pointer (see Item 28) that automatically deletes what it points to when the pointer itself is destroyed. In fact, the standard C++ library contains a template, `auto_ptr`, for just such a smart pointer (see Item 9). By making `collisionMap` a static `auto_ptr` in `lookup`, we can have `initializeCollisionMap` return a pointer to an initialized `map` object, yet never have to worry about a resource leak; the `map` to which `collisionMap` points will be automatically destroyed when `collisionMap` is. Thus:

Click here to view code image

```
class SpaceShip: public GameObject {
private:
  static HitMap * initializeCollisionMap();
  ...

};

SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static auto_ptr<HitMap>
    collisionMap(initializeCollisionMap());
  ...

}
```

The clearest way to implement `initializeCollisionMap` would seem to be this,

Click here to view code image

```
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;

  (*phm)["SpaceShip"] = &hitSpaceShip;
  (*phm)["SpaceStation"] = &hitSpaceStation;
  (*phm)["Asteroid"] = &hitAsteroid;

  return phm;
}
```

but as I noted earlier, this won't compile. That's because a `HitMap` is declared to hold pointers to member functions that all take the same type of argument, namely `GameObject`. But `hitSpaceShip` takes a `SpaceShip`, `hitSpaceStation` takes a `SpaceStation`, and, `hitAsteroid` takes an `Asteroid`. Even though `SpaceShip`, `SpaceStation`, and `Asteroid` can all be implicitly converted to `GameObject`, there is no such conversion for pointers to functions taking these argument types.

To placate your compilers, you might be tempted to employ `reinterpret_cast`s (see Item 2), which are generally the casts of choice when converting between function pointer types:

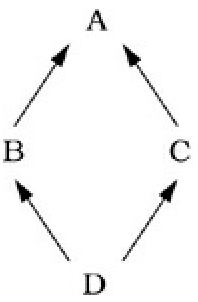
Click here to view code image

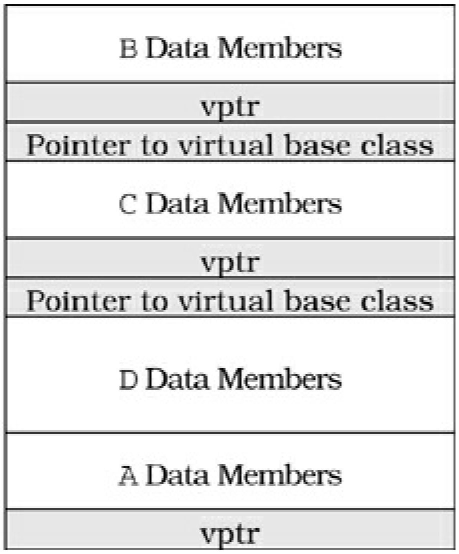
```
// A bad idea...
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;

  (*phm)["SpaceShip"] =
    reinterpret_cast<HitFunctionPtr>(&hitSpaceShip);

  (*phm)["SpaceStation"] =
    reinterpret_cast<HitFunctionPtr>(&hitSpaceStation);

  (*phm)["Asteroid"] =
    reinterpret_cast<HitFunctionPtr>(&hitAsteroid);

  return phm;
}
```

This will compile, but it's a bad idea. It entails doing something you should never do: lying to your compilers. Telling them that `hitSpaceShip`, `hitSpaceStation`, and `hitAsteroid` are functions expecting a `GameObject` argument is simply not true. `hitSpaceShip` expects a `SpaceShip`, `hitSpaceStation` expects a `SpaceStation`, and `hitAsteroid` expects an `Asteroid`. The casts say otherwise. The casts lie.

More than morality is on the line here. Compilers don't like to be lied to, and they often find a way to exact revenge when they discover they've been deceived. In this case, they're likely to get back at you by generating bad code for functions you call through `*phm` in cases where `GameObject`'s derived classes employ multiple inheritance or have virtual base classes. In other words, if `SpaceStation`, `SpaceShip`, or `Asteroid` had other base classes (in addition to `GameObject`), you'd probably find that your calls to collision-processing functions in `collide` would behave quite rudely.

Consider again the A-B-C-D inheritance hierarchy and the possible object layout for a `D` object that is described in Item 24:





Each of the four class parts in a `D` object has a different address. This is important, because even though pointers and references behave differently (see Item 1), compilers typically *implement* references by using pointers in the generated code. Thus, pass-by-reference is typically implemented by passing a pointer to an object. When an object with multiple base classes (such as a `D` object) is passed by reference, it is crucial that compilers pass the *correct* address — the one corresponding to the declared type of the parameter in the function being called.

But what if you've lied to your compilers and told them your function expects a `GameObject` when it really expects a `SpaceShip` or a `SpaceStation`? Then they'll pass the wrong address when you call the function, and the resulting runtime carnage will probably be gruesome. It will also be *very* difficult to determine the cause of the problem. There are good reasons why casting is discouraged. This is one of them.

Okay, so casting is out. Fine. But the type mismatch between the function pointers a `HitMap` is willing to contain and the pointers to the `hitSpaceShip`, `hitSpaceStation`, and `hitAsteroid` functions remains. There is only one way to resolve the conflict: change the types of the functions so they all take `GameObject` arguments:

Click here to view code image

```
class GameObject {                    // this is unchanged
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};
class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);

  // these functions now all take a GameObject parameter
  virtual void hitSpaceShip(GameObject& spaceShip);
  virtual void hitSpaceStation(GameObject& spaceStation);
  virtual void hitAsteroid(GameObject& asteroid);
  ...
};
```

Our solution to the double-dispatching problem that was based on virtual functions overloaded the function name `collide`. Now we are in a position to understand why we didn't follow suit here — why we decided to use an associative array of member function pointers instead. All the `hit` functions take the same parameter type, so we must give them different names.

Now we can write `initializeCollisionMap` the way we always wanted to:

Click here to view code image

```
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;
  (*phm)["SpaceShip"] = &hitSpaceShip;
  (*phm)["SpaceStation"] = &hitSpaceStation;
  (*phm)["Asteroid"] = &hitAsteroid;

  return phm;
}
```

Regrettably, our `hit` functions now get a general `GameObject` parameter instead of the derived class parameters they expect. To bring reality into accord with expectation, we must resort to a `dynamic_cast` (see Item 2) at the top of each function:

Click here to view code image

```
void SpaceShip::hitSpaceShip(GameObject& spaceShip)
{
  SpaceShip& otherShip=
    dynamic_cast<SpaceShip&>(spaceShip);

  process a SpaceShip-SpaceShip collision;
}

void SpaceShip::hitSpaceStation(GameObject& spaceStation)
{
  SpaceStation& station=
    dynamic_cast<SpaceStation&>(spaceStation);

  process a SpaceShip-SpaceStation collision;
}
void SpaceShip::hitAsteroid(GameObject& asteroid)
{
  Asteroid& theAsteroid =
    dynamic_cast<Asteroid&>(asteroid);

  process a SpaceShip-Asteroid collision;
}
```

Each of the `dynamic_cast`s will throw a `bad_cast` exception if the cast fails. They should never fail, of course, because the `hit` functions should never be called with incorrect parameter types. Still, we're better off safe than sorry.

Using Non-Member Collision-Processing Functions

We now know how to build a vtbl-like associative array that lets us implement the second half of a double-dispatch, and we know how to encapsulate the details of the associative array inside a lookup function. Because this array contains pointers to *member* functions, however, we still have to modify class definitions if a new type of `GameObject` is added to the game, and that means everybody has to recompile, even people who don't care about the new type of object. For example, if `Satellite` were added to our game, we'd have to augment the `SpaceShip` class with a declaration of a function to handle collisions between satellites and spaceships. All `SpaceShip` clients would then have to recompile, even if they couldn't care less about the existence of satellites. This is the problem that led us to reject the implementation of double-dispatching based purely on virtual functions, and that solution was a lot less work than the one we've just seen.

The recompilation problem would go away if our associative array contained pointers to non-member functions. Furthermore, switching to non-member collision-processing functions would let us address a design question we have so far ignored, namely, in which class should collisions between objects of different types be handled? With the implementation we just developed, if object 1 and object 2 collide and object 1 happens to be the left-hand argument to `processCollision`, the collision will be handled inside the class for object 1. If object 2 happens to be the left-hand argument to `processCollision`, however, the collision will be handled inside the class for object 2. Does this make sense? Wouldn't it be better to design things so that collisions between objects of types A and B are handled by neither A nor B but instead in some neutral location outside both classes?

If we move the collision-processing functions out of our classes, we can give clients header files that contain class definitions without any `hit` or `collide` functions. We can then structure our implementation file for `processCollision` as follows:

Click here to view code image

```
#include "SpaceShip.h"
#include "SpaceStation.h"
#include "Asteroid.h"

namespace {                 // unnamed namespace — see below

  // primary collision-processing functions
  void shipAsteroid(GameObject& spaceShip,
                    GameObject& asteroid);
  void shipStation(GameObject& spaceShip,
                   GameObject& spaceStation);

  void asteroidStation(GameObject& asteroid,
                       GameObject& spaceStation);
   ...

  // secondary collision-processing functions that just
  // implement symmetry: swap the parameters and call a
  // primary function
  void asteroidShip(GameObject& asteroid,
                    GameObject& spaceShip)
  { shipAsteroid(spaceShip, asteroid); }

  void stationShip(GameObject& spaceStation,
                   GameObject& spaceShip)
  { shipStation(spaceShip, spaceStation); }

  void stationAsteroid(GameObject& spaceStation,
                       GameObject& asteroid)
  { asteroidStation(asteroid, spaceStation); }

  ...

  // see below for a description of these types/functions
  typedef void (*HitFunctionPtr)(GameObject&, GameObject&);
  typedef map< pair<string,string>, HitFunctionPtr >
HitMap;

  pair<string,string> makeStringPair(const char *s1,
                                     const char *s2);
  HitMap * initializeCollisionMap();

  HitFunctionPtr lookup(const string& class1,
                        const string& class2);
} // end namespace

void processCollision(GameObject& object1,
                      GameObject& object2)
{
  HitFunctionPtr phf = lookup(typeid(object1).name(),
                              typeid(object2).name());
  if (phf) phf(object1, object2);
  else throw UnknownCollision(object1, object2);
```

```
}
```

Note the use of the unnamed namespace to contain the functions used to implement `processCollision`. Everything in such an unnamed namespace is private to the current translation unit (essentially the current file) — it's just like the functions were declared `static` at file scope. With the advent of namespaces, however, statics at file scope have been deprecated, so you should accustom yourself to using unnamed namespaces as soon as your compilers support them.

Conceptually, this implementation is the same as the one that used member functions, but there are some minor differences. First, `HitFunctionPtr` is now a typedef for a pointer to a non-member function. Second, the exception class `CollisionWithUnknownObject` has been renamed `UnknownCollision` and modified to take two objects instead of one. Finally, `lookup` must now take two type names and perform both parts of the double-dispatch. This means our collision map must now hold three pieces of information: two types names and a `HitFunctionPtr`.

As fate would have it, the standard `map` class is defined to hold only two pieces of information. We can finesse that problem by using the standard `pair` template, which lets us bundle the two type names together as a single object. `initializeCollisionMap`, along with its `makeStringPair` helper function, then looks like this:

Click here to view code image

```
// we use this function to create pair<string,string>
// objects from two char* literals. It's used in
// initializeCollisionMap below. Note how this function
// enables the return value optimization (see Item 20).

namespace {       // unnamed namespace again - see below

  pair<string,string> makeStringPair(const char *s1,
                                     const char *s2)
  { return pair<string,string>(s1, s2); }

} // end namespace

namespace {       // still the unnamed namespace - see below

  HitMap * initializeCollisionMap()
  {
    HitMap *phm = new HitMap;

    (*phm)[makeStringPair("SpaceShip","Asteroid")] =
      &shipAsteroid;

    (*phm)[makeStringPair("SpaceShip", "SpaceStation")] =
      &shipStation;

     ...

     return phm;
  }

} // end namespace
```

`lookup` must also be modified to work with the `pair<string, string>` objects that now comprise the first component of the collision map:

[Click here to view code image]

```
namespace {       // I explain this below - trust me

  HitFunctionPtr lookup(const string& class1,
                        const string& class2)
  {
    static auto_ptr<HitMap>
      collisionMap(initializeCollisionMap());

    // see below for a description of make_pair
    HitMap::iterator mapEntry=
      collisionMap->find(make_pair(class1, class2));

    if (mapEntry == collisionMap->end()) return 0;

    return (*mapEntry).second;
  }

} // end namespace
```

This is almost exactly what we had before. The only real difference is the use of the `make_pair` function in this statement:

[Click here to view code image]

```
HitMap::iterator mapEntry=
  collisionMap->find(make_pair(class1, class2));
```

`make_pair` is just a convenience function (template) in the standard library (see Item 35) that saves us the trouble of specifying the types when constructing a `pair` object. We could just as well have written the statement like this:

Click here to view code image

```
HitMap::iterator mapEntry=
  collisionMap->find(pair<string,string>(class1, class2));
```

This calls for more typing, however, and specifying the types for the `pair` is redundant (they're the same as the types of `class1` and `class2`), so the `make_pair` form is more commonly used.

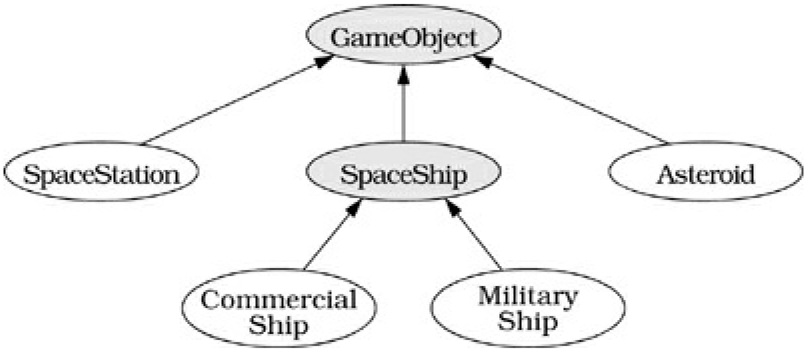
Because `makeStringPair`, `initializeCollisionMap`, and `lookup` were declared inside an unnamed namespace, each must be implemented within the same namespace. That's why the implementations of the functions above are in the unnamed namespace (for the same translation unit as their declarations): so the linker will correctly associate their definitions (i.e., their implementations) with their earlier declarations.

We have finally achieved our goals. If new subclasses of `GameObject` are added to our hierarchy, existing classes need not recompile (unless they wish to use the new classes). We have no tangle of RTTI-based `switch` or `if-then-else` conditionals to maintain. The addition of new classes to the hierarchy requires only well-defined and localized changes to our system: the addition of one or more map insertions in `initializeCollisionMap` and the declarations of the new collision-processing functions in the unnamed namespace associated with the implementation of `processCollision`. It may have been a lot of work to get here, but at least the trip was worthwhile. Yes? Yes?

Maybe.

Inheritance and Emulated Virtual Function Tables

There is one final problem we must confront. (If, at this point, you are wondering if there will *always* be one final problem to confront, you have truly come to appreciate the difficulty of designing an implementation mechanism for virtual functions.) Everything we've done will work fine as long as we never need to allow inheritance-based type conversions when calling collision-processing functions. But suppose we develop a game in which we must sometimes distinguish between commercial space ships and military space ships. We could modify our hierarchy as follows, where we've heeded the guidance of Item 33 and made the concrete classes `CommercialShip` and `MilitaryShip` inherit from the newly abstract class `SpaceShip`:



Suppose commercial and military ships behave identically when they collide with something. Then we'd expect to be able to use the same collision-processing functions we had before `CommercialShip` and `MilitaryShip` were added. In particular, if a `MilitaryShip` object and an `Asteroid` collided, we'd expect

Click here to view code image

```
void shipAsteroid(GameObject& spaceShip,
                  GameObject& asteroid);
```

to be called. It would not be. Instead, an `UnknownCollision` exception would be thrown. That's because `lookup` would be asked to find a function corresponding to the type names "MilitaryShip" and "Asteroid," and no such function would be found in `collisionMap`. Even though a `MilitaryShip` can be treated like a `SpaceShip`, `lookup` has no way of knowing that.

Furthermore, there is no easy way of telling it. If you need to implement double-dispatching and you need to support inheritance-based parameter conversions such as these, your only practical recourse is to fall back on the double-virtual-function-call mechanism we examined earlier. That implies you'll also have to put up with everybody recompiling when you add to your inheritance hierarchy, but that's just the way life is sometimes.

Initializing Emulated Virtual Function Tables (Reprise)

That's really all there is to say about double-dispatching, but it would be unpleasant to end the discussion on such a downbeat note, and unpleasantness is, well, unpleasant. Instead, let's conclude by outlining an alternative approach to initializing `collisionMap`.

As things stand now, our design is entirely static. Once we've registered a function for processing collisions between two types of objects, that's it; we're stuck with that function forever. What if we'd like to add, remove, or change collision-processing functions as the game proceeds? There's no way to do it.

But there can be. We can turn the concept of a map for storing collision-processing functions into a class that offers member functions allowing us to modify the contents of the map dynamically. For example:

Click here to view code image

```
class CollisionMap {
public:
  typedef void (*HitFunctionPtr)(GameObject&, GameObject&);

  void addEntry(const string& type1,
                const string& type2,
                HitFunctionPtr collisionFunction,
                bool symmetric = true);        // see below

  void removeEntry(const string& type1,
                   const string& type2);

  HitFunctionPtr lookup(const string& type1,
                        const string& type2);

  // this function returns a reference to the one and only
  // map - see Item 26
  static CollisionMap& theCollisionMap();

private:
  // these functions are private to prevent the creation
  // of multiple maps - see Item 26
  CollisionMap();
  CollisionMap(const CollisionMap&);
};
```

This class lets us add entries to the map, remove them from it, and look up the collision-processing function associated with a particular pair of type names. It also uses the techniques of Item 26 to limit the number of

`CollisionMap` objects to one, because there is only one map in our system. (More complex games with multiple maps are easy to imagine.) Finally, it allows us to simplify the addition of symmetric collisions to the map (i.e., collisions in which the effect of an object of type T1 hitting an object of type T2 is the same as that of an object of type T2 hitting an object of type T1) by automatically adding the implied map entry when `addEntry` is called with the optional parameter `symmetric` set to `true`.

With the `CollisionMap` class, each client wishing to add an entry to the map does so directly:

Click here to view code image

```
void shipAsteroid(GameObject& spaceShip,
                  GameObject& asteroid);
CollisionMap::theCollisionMap().addEntry("SpaceShip",
                                         "Asteroid",
                                         &shipAsteroid);

void shipStation(GameObject& spaceShip,
                 GameObject& spaceStation);
CollisionMap::theCollisionMap().addEntry("SpaceShip",
                                         "SpaceStation",
                                         &shipStation);
void asteroidStation(GameObject& asteroid,
                     GameObject& spaceStation);
CollisionMap::theCollisionMap().addEntry("Asteroid",
                                         "SpaceStation",
                                         &asteroidStation);
...
```

Care must be taken to ensure that these map entries are added to the map before any collisions occur that would call the associated functions. One way to do this would be to have constructors in `GameObject` subclasses check to make sure the appropriate mappings had been added each time an object was created. Such an approach would exact a small performance penalty at runtime. An alternative would be to create a `RegisterCollisionFunction` class:

Click here to view code image

```
class RegisterCollisionFunction {
public:
  RegisterCollisionFunction(
            const string& type1,
            const string& type2,
            CollisionMap::HitFunctionPtr collisionFunction,
            bool symmetric = true)
  {
    CollisionMap::theCollisionMap().addEntry(type1, type2,

collisionFunction,
                                             symmetric);
  }
};
```

Clients could then use global objects of this type to automatically register the functions they need:

Click here to view code image

```
RegisterCollisionFunction cf1("SpaceShip", "Asteroid",
                              &shipAsteroid);

RegisterCollisionFunction cf2("SpaceShip", "SpaceStation",
                              &shipStation);

RegisterCollisionFunction cf3("Asteroid", "SpaceStation",
                              &asteroidStation);
...

int main(int argc, char * argv[])
{
  ...
}
```

Because these objects are created before `main` is invoked, the functions their constructors register are also added to the map before `main` is called. If, later, a new derived class is added

```
class Satellite: public GameObject { ... };
```

and one or more new collision-processing functions are written,

Click here to view code image

```
void satelliteShip(GameObject& satellite,
                   GameObject& spaceShip);

void satelliteAsteroid(GameObject& satellite,
                       GameObject& asteroid);
```

these new functions can be similarly added to the map without disturbing existing code:

[Click here to view code image](#)

```
RegisterCollisionFunction cf4("Satellite", "SpaceShip",
                              &satelliteShip);

RegisterCollisionFunction cf5("Satellite", "Asteroid",
                              &satelliteAsteroid);
```

This doesn't change the fact that there's no perfect way to implement multiple dispatch, but it does make it easy to provide data for a map-based implementation if we decide such an approach is the best match for our needs.

Miscellany

We thus arrive at the organizational back of the bus, the chapter containing the guidelines no one else would have. We begin with two Items on C++ software development that describe how to design systems that accommodate change. One of the strengths of the object-oriented approach to systems building is its support for change, and these Items describe specific steps you can take to fortify your software against the slings and arrows of a world that refuses to stand still.

We then examine how to combine C and C++ in the same program. This necessarily leads to consideration of extralinguistic issues, but C++ exists in the real world, so sometimes we must confront such things.

Finally, I summarize changes to the C++ language standard since publication of the *de facto* reference. I especially cover the sweeping changes that have been made in the standard library. If you have not been following the standardization process closely, you are probably in for some surprises — many of them quite pleasant.

Item 32: Program in the future tense

Things change.

As software developers, we may not know much, but we do know that things will change. We don't necessarily know what will change, how the changes will be brought about, when the changes will occur, or why they will take place, but we do know this: things will change.

Good software adapts well to change. It accommodates new features, it ports to new platforms, it adjusts to new demands, it handles new inputs. Software this flexible, this robust, and this reliable does not come about by accident. It is designed and implemented by programmers who conform to the constraints of today while keeping in mind the probable needs of tomorrow. This kind of software — software that accepts change gracefully — is written by people who *program in the future tense*.

To program in the future tense is to accept that things will change and to be prepared for it. It is to recognize that new functions will be added to libraries, that new overloadings will occur, and to watch for the potentially ambiguous function calls that might result. It is to acknowledge that new classes will be added to hierarchies, that present-day derived classes may be tomorrow's base classes, and to prepare for that possibility. It is to accept that new applications will be written, that functions will be called in new contexts, and to write those functions so they continue to perform correctly. It is to remember that the programmers charged with software maintenance are typically not the code's original developers, hence to design and implement in a fashion that facilitates comprehension, modification, and enhancement by others.

One way to do this is to express design constraints in C++ instead of (or in addition to) comments or other documentation. For example, if a class is designed to never have derived classes, don't just put a comment in the header file above the class, use C++ to prevent derivation; Item 26 shows you how. If a class requires that all instances be on the heap, don't just tell clients that, enforce the restriction by applying the approach of Item 27. If copying and assignment make no sense for a class, prevent those operations by declaring the copy constructor and the assignment operator private. C++ offers great power, flexibility, and expressiveness. Use these characteristics of the language to enforce the design decisions in your programs.

Given that things will change, write classes that can withstand the rough-and-tumble world of software evolution. Avoid "demand-paged" virtual functions, whereby you make no functions virtual unless somebody comes along and demands that you do it. Instead, determine the *meaning* of a function and whether it makes sense to let it be redefined in derived classes. If it does, declare it virtual, even if nobody redefines it right away. If it doesn't, declare it nonvirtual, and don't change it later just because it would be convenient for someone; make sure the change makes sense in the context of the entire class and the abstraction it represents.

Handle assignment and copy construction in every class, even if "nobody ever does those things." Just because they don't do them now doesn't mean they won't do them in the future. If these functions are difficult to implement, declare them `private`. That way no one will inadvertently call compiler-generated functions that do the wrong thing (as often happens with default assignment operators and copy constructors).

Adhere to the principle of least astonishment: strive to provide classes whose operators and functions have a natural syntax and an intuitive semantics. Preserve consistency with the behavior of the built-in types: when in doubt, do as the `int`s do.

Recognize that anything somebody *can* do, they *will* do. They'll throw exceptions, they'll assign objects to themselves, they'll use objects before giving them values, they'll give objects values and never use them, they'll give them huge values, they'll give them tiny values, they'll give them null values. In general, if it will compile, somebody will do it. As a result, make your classes easy to use correctly and hard to use incorrectly. Accept that clients will make mistakes, and design your classes so you can prevent, detect, or correct such errors (see, for example, Item 33).

Strive for portable code. It's not much harder to write portable programs than to write unportable ones, and only rarely will the difference in performance be significant enough to justify unportable constructs (see Item 16). Even programs designed for custom hardware often end up being ported, because stock hardware generally achieves an equivalent level of performance within a few years. Writing portable code allows you to switch platforms easily, to enlarge your client base, and to brag about supporting open systems. It also makes it easier to recover if you bet wrong in the operating system sweepstakes.

Design your code so that when changes are necessary, the impact is localized. Encapsulate as much as you can; make implementation details private. Where applicable, use unnamed namespaces or file-`static` objects and functions (see Item 31). Try to avoid designs that lead to virtual base classes, because such classes must be initialized by every class derived from them — even those derived indirectly (see Item 4). Avoid RTTI-based designs that make use of cascading `if-then-else` statements (see Item 31 again). Every time the class hierarchy changes, each set of statements must be updated, and if you forget one, you'll receive no warning from your compilers.

These are well known and oft-repeated exhortations, but most programmers are still stuck in the present tense. As are many authors, unfortunately. Consider this advice by a well-regarded C++ expert:

> You need a virtual destructor whenever someone deletes a `B*` that actually

points to a `D`.

Here `B` is a base class and `D` is a derived class. In other words, this author suggests that if your program looks like this, you don't need a virtual destructor in `B`:

Click here to view code image

```
class B { ... };                // no virtual dtor needed
class D: public B { ... };

B *pb = new D;
```

However, the situation changes if you add this statement:

Click here to view code image

```
delete pb;                      // NOW you need the virtual
                                // destructor in B
```

The implication is that a minor change to client code — the addition of a `delete` statement — can result in the need to change the class definition for `B`. When that happens, all `B`'s clients must recompile. Following this author's advice, then, the addition of a single statement in one function can lead to extensive code recompilation and relinking for all clients of a library. This is anything but effective software design.

On the same topic, a different author writes:

If a public base class does not have a virtual destructor, no derived class nor members of a derived class should have a destructor.

In other words, this is okay,

Click here to view code image

```
class string {                // from the standard C++
library
public:
  ~string();
};

class B { ... };              // no data members with
dtors,
                              // no virtual dtor needed
```

but if a new class is derived from `B`, things change:

Click here to view code image

```
class D: public B {
  string name;                // NOW ~B needs to be virtual
};
```

Again, a small change to the way `B` is used (here, the addition of a derived class that contains a member with a destructor) may necessitate extensive recompilation and relinking by clients. But small changes in software should have small impacts on systems. This design fails that test.

The same author writes:

> If a multiple inheritance hierarchy has any destructors, every base class should have a virtual destructor.

In all these quotations, note the present-tense thinking. How do clients manipulate pointers *now*? What class members have destructors *now*? What classes in the hierarchy have destructors *now*?

Future-tense thinking is quite different. Instead of asking how a class is used now, it asks how the class is *designed* to be used. Future-tense thinking says, if a class is *designed* to be used as a base class (even if it's not used as one now), it should have a virtual destructor. Such classes behave correctly both now and in the future, and they don't affect other library clients when new classes derive from them. (At least, they have no effect as far as their destructor is concerned. If additional changes to the class are required, other clients may be affected.)

A commercial class library (one that predates the `string` specification in the C++ library standard) contains a string class with no virtual destructor. The vendor's explanation?

> We didn't make the destructor virtual, because we didn't want `String` to have a vtbl. We have no intention of ever having a `String*`, so this is not a problem. We are well aware of the difficulties this could cause.

Is this present-tense or future-tense thinking?

Certainly the vtbl issue is a legitimate technical concern (see Item 24). The implementation of most `String` classes contains only a single `char*` pointer inside each `String` object, so adding a vptr to each `String` would double the size of those objects. It is easy to understand why a vendor would be unwilling to do that, especially for a highly visible, heavily used class like `String`. The performance of such a class might easily fall within the 20% of a program that makes a difference (see Item 16).

Still, the total memory devoted to a string object — the memory for the object itself plus the heap memory needed to hold the string's value — is typically much greater than just the space needed to hold a `char*` pointer. From this perspective, the overhead imposed by a vptr is less significant. Nevertheless, it is a legitimate technical consideration. (Certainly the ISO/ANSI standardization committee seems to think so: the standard `string` type has a nonvirtual destructor.)

Somewhat more troubling is the vendor's remark, "We have no intention of ever having a `String*`, so this is not a problem." That may be true, but their `String` class is part of a library they make available to *thousands* of developers. That's a lot of developers, each with a different level of experience with C++, each doing something unique. Do those developers understand the consequences of there being no virtual destructor in `String`? Are they likely to know that because `String` has no virtual destructor, deriving new classes from `String` is a high-risk venture? Is this vendor confident their clients will understand that in the absence of a virtual destructor, deleting objects through `String*` pointers will not work properly and RTTI operations on pointers and references to `String`s may return incorrect information? Is this class easy to use correctly and hard to use incorrectly?

This vendor should provide documentation for its `String` class that makes clear the class is not designed for derivation, but what if programmers overlook the caveat or flat-out fail to read the documentation?

An alternative would be to use C++ itself to prohibit derivation. Item 26 describes how to do this by limiting object creation to the heap and then using `auto_ptr` objects to manipulate the heap objects. The interface for `String` creation would then be both unconventional and inconvenient, requiring this,

Click here to view code image

```
auto_ptr<String> ps(String::makeString("Future tense
C++"));
```

```
  ...                              // treat ps as a pointer to
                                   // a String object, but
  don't
                                   // worry about deleting it
```

instead of this,

```
  String s("Future tense C++");
```

but perhaps the reduction in the risk of improperly behaving derived classes would be worth the syntactic inconvenience. (For `String`, this is unlikely to be the case, but for other classes, the trade-off might well be worth it.)

There is a need, of course, for present-tense thinking. The software you're developing has to work with current compilers; you can't afford to wait until the latest language features are implemented. It has to run on the hardware you currently support and it must do so under configurations your clients have available; you can't force your customers to upgrade their systems or modify their operating environment. It has to offer acceptable performance *now*; promises of smaller, faster programs some years down the line don't generally warm the cockles of potential customers' hearts. And the software you're working on must be available "soon," which often means some time in the recent past. These are important constraints. You cannot ignore them.

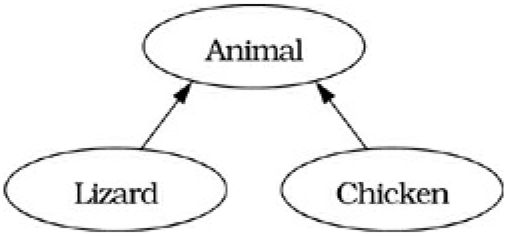
Future-tense thinking simply adds a few additional considerations:

- Provide complete classes, even if some parts aren't currently used. When new demands are made on your classes, you're less likely to have to go back and modify them.
- Design your interfaces to facilitate common operations and prevent common errors. Make the classes easy to use correctly, hard to use incorrectly. For example, prohibit copying and assignment for classes where those operations make no sense. Prevent partial assignments (see Item 33).
- If there is no great penalty for generalizing your code, generalize it. For example, if you are writing an algorithm for tree traversal, consider generalizing it to handle any kind of directed acyclic graph.

Future tense thinking increases the reusability of the code you write, enhances its maintainability, makes it more robust, and facilitates graceful change in an environment where change is a certainty. It must be balanced against present-tense constraints. Too many programmers focus exclusively on current needs, however, and in doing so they sacrifice the long-term viability of the software they design and implement. Be different. Be a renegade. Program in the future tense.

Item 33: Make non-leaf classes abstract

Suppose you're working on a project whose software deals with animals. Within this software, most animals can be treated pretty much the same, but two kinds of animals — lizards and chickens — require special handling. That being the case, the obvious way to relate the classes for animals, lizards, and chickens is like this:



The `Animal` class embodies the features shared by all the creatures you deal with, and the `Lizard` and `Chicken` classes specialize `Animal` in ways appropriate for lizards and chickens, respectively.

Here's a sketch of the definitions for these classes:

[Click here to view code image]()

```
class Animal {
public:
  Animal& operator=(const Animal& rhs);
  ...

};
class Lizard: public Animal {
public:
  Lizard& operator=(const Lizard& rhs);
  ...

};
```

```
class Chicken: public Animal {
public:
  Chicken& operator=(const Chicken& rhs);
  ...

};
```

Only the assignment operators are shown here, but that's more than enough to keep us busy for a while. Consider this code:

Click here to view code image

```
Lizard liz1;
Lizard liz2;

Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;

...

*pAnimal1 = *pAnimal2;
```

There are two problems here. First, the assignment operator invoked on the last line is that of the `Animal` class, even though the objects involved are of type `Lizard`. As a result, only the `Animal` part of `liz1` will be modified. This is a *partial* assignment. After the assignment, `liz1`'s `Animal` members have the values they got from `liz2`, but `liz1`'s `Lizard` members remain unchanged.

The second problem is that real programmers write code like this. It's not uncommon to make assignments to objects through pointers, especially for experienced C programmers who have moved to C++. That being the case, we'd like to make the assignment behave in a more reasonable fashion. As Item 32 points out, our classes should be easy to use correctly and difficult to use incorrectly, and the classes in the hierarchy above are easy to use incorrectly.

One approach to the problem is to make the assignment operators virtual. If `Animal::operator=` were virtual, the assignment would invoke the `Lizard` assignment operator, which is certainly the correct one to call. However, look what happens if we declare the assignment operators virtual:

Click here to view code image

```
class Animal {
public:
  virtual Animal& operator=(const Animal& rhs);
  ...
};
class Lizard: public Animal {
public:
  virtual Lizard& operator=(const Animal& rhs);
  ...
};

class Chicken: public Animal {
public:
  virtual Chicken& operator=(const Animal& rhs);
  ...
};
```

Due to relatively recent changes to the language, we can customize the return value of the assignment operators so that each returns a reference to the correct class, but the rules of C++ force us to declare identical *parameter* types for a virtual function in every class in which it is declared. That means the assignment operator for the `Lizard` and `Chicken` classes must be prepared to accept *any* kind of `Animal` object on the right-hand side of an assignment. That, in turn, means we have to confront the fact that code like the following is legal:

Click here to view code image

```
Lizard liz;
Chicken chick;

Animal *pAnimal1 = &liz;
Animal *pAnimal2 = &chick;

...

*pAnimal1 = *pAnimal2;          // assign a chicken to
                                // a lizard!
```

This is a mixed-type assignment: a `Lizard` is on the left and a `Chicken` is on the right. Mixed-type assignments aren't usually a problem in C++, because the language's strong typing generally renders them illegal. By making `Animal`'s assignment operator virtual, however, we opened the door to such mixed-type operations.

This puts us in a difficult position. We'd like to allow same-type assignments through pointers, but we'd like to forbid mixed-type assignments through those same pointers. In other words, we want to allow this,

Click here to view code image

```
Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;

...

*pAnimal1 = *pAnimal2;        // assign a lizard to a
lizard
```

but we want to prohibit this:

Click here to view code image

```
Animal *pAnimal1 = &liz;
Animal *pAnimal2 = &chick;

...

*pAnimal1 = *pAnimal2;        // assign a chicken to a
lizard
```

Distinctions such as these can be made only at runtime, because sometimes assigning `*pAnimal2` to `*pAnimal1` is valid, sometimes it's not. We thus enter the murky world of type-based runtime errors. In particular, we need to signal an error inside `operator=` if we're faced with a mixed-type assignment, but if the types are the same, we want to perform the assignment in the usual fashion.

We can use a `dynamic_cast` (see Item 2) to implement this behavior. Here's how to do it for `Lizard`'s assignment operator:

Click here to view code image

```
Lizard& Lizard::operator=(const Animal& rhs)
{
  // make sure rhs is really a lizard
  const Lizard& rhs_liz = dynamic_cast<const Lizard&>(rhs);

  proceed with a normal assignment of rhs_liz to *this;
}
```

This function assigns `rhs` to `*this` only if `rhs` is really a `Lizard`. If it's not, the function propagates the `bad_cast` exception that `dynamic_cast` throws when the cast to a reference fails. (Actually, the type of the exception is `std::bad_cast`, because the components of the standard library, including the exceptions thrown by the standard components, are in the namespace `std`. For an overview of the standard library, see Item 35.)

Even without worrying about exceptions, this function seems needlessly complicated and expensive — the `dynamic_cast` must consult a `type_info` structure; see Item 24 — in the common case where one `Lizard` object is assigned to another:

Click here to view code image

```
Lizard liz1, liz2;

...

liz1 = liz2;                // no need to perform a
                            // dynamic_cast: this
                            // assignment must be valid
```

We can handle this case without paying for the complexity or cost of a `dynamic_cast` by adding to `Lizard` the conventional assignment operator:

Click here to view code image

```
class Lizard: public Animal {
public:
  virtual Lizard& operator=(const Animal& rhs);

  Lizard& operator=(const Lizard& rhs);        // add this

  ...

};

Lizard liz1, liz2;

...

liz1 = liz2;                     // calls operator= taking
                                 // a const Lizard&

Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;

...
```

```
*pAnimal1 = *pAnimal2;          // calls operator= taking
                                // a const Animal&
```

In fact, given this latter `operator=`, it's simplicity itself to implement the former one in terms of it:

Click here to view code image

```
Lizard& Lizard::operator=(const Animal& rhs)
{
  return operator=(dynamic_cast<const Lizard&>(rhs));
}
```

This function attempts to cast `rhs` to be a `Lizard`. If the cast succeeds, the normal class assignment operator is called. Otherwise, a `bad_cast` exception is thrown.

Frankly, all this business of checking types at runtime and using `dynamic_cast`s makes me nervous. For one thing, some compilers still lack support for `dynamic_cast`, so code that uses it, though theoretically portable, is not necessarily portable in practice. More importantly, it requires that clients of `Lizard` and `Chicken` be prepared to catch `bad_cast` exceptions and do something sensible with them each time they perform an assignment. In my experience, there just aren't that many programmers who are willing to program that way. If they don't, it's not clear we've gained a whole lot over our original situation where we were trying to guard against partial assignments.

Given this rather unsatisfactory state of affairs regarding virtual assignment operators, it makes sense to regroup and try to find a way to prevent clients from making problematic assignments in the first place. If such assignments are rejected during compilation, we don't have to worry about them doing the wrong thing.

The easiest way to prevent such assignments is to make `operator=` private in `Animal`. That way, lizards can be assigned to lizards and chickens can be assigned to chickens, but partial and mixed-type assignments are forbidden:

Click here to view code image

```
class Animal {
private:
  Animal& operator=(const Animal& rhs);      // this is now
  ...                                        // private
};

class Lizard: public Animal {
public:
  Lizard& operator=(const Lizard& rhs);
  ...

};

class Chicken: public Animal {
public:
  Chicken& operator=(const Chicken& rhs);
  ...

};

Lizard liz1, liz2;
...
liz1 = liz2;                        // fine

Chicken chick1, chick2;
...
chick1 = chick2;                    // also fine

Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &chick1;
...
*pAnimal1 = *pAnimal2;              // error! attempt to call
                                    // private
Animal::operator=
```

Unfortunately, `Animal` is a concrete class, and this approach also makes assignments between `Animal` objects illegal:

Click here to view code image

```
Animal animal1, animal2;

...

animal1 = animal2;                  // error! attempt to call
                                    // private
Animal::operator=
```

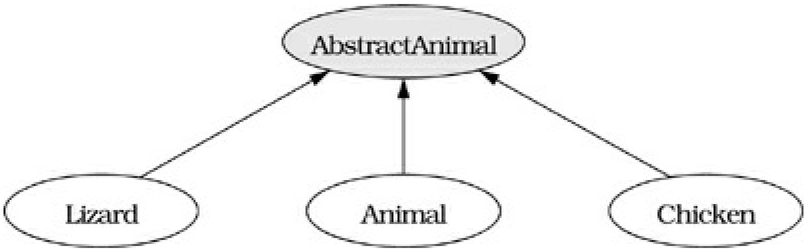
Moreover, it makes it impossible to implement the `Lizard` and `Chicken` assignment operators correctly, because assignment operators in derived classes are responsible for calling assignment operators in their base classes:

[Click here to view code image]

```
Lizard& Lizard::operator=(const Lizard& rhs)
{
  if (this == &rhs) return *this;

  Animal::operator=(rhs);         // error! attempt to call
                                  // private function. But
                                  // Lizard::operator= must
                                  // call this function to
  ...                             // assign the Animal parts
}                                 // of *this!
```

We can solve this latter problem by declaring `Animal::operator=` `protected`, but the conundrum of allowing assignments between `Animal` objects while preventing partial assignments of `Lizard` and `Chicken` objects through `Animal` pointers remains. What's a poor programmer to do?

The easiest thing is to eliminate the need to allow assignments between `Animal` objects, and the easiest way to do that is to make `Animal` an abstract class. As an abstract class, `Animal` can't be instantiated, so there will be no need to allow assignments between `Animal`s. Of course, this leads to a new problem, because our original design for this system presupposed that `Animal` objects were necessary. There is an easy way around this difficulty. Instead of making `Animal` itself abstract, we create a new class — `AbstractAnimal`, say — consisting of the common features of `Animal`, `Lizard`, and `Chicken` objects, and we make *that* class abstract. Then we have each of our concrete classes inherit from `AbstractAnimal`. The revised hierarchy looks like this,



and the class definitions are as follows:

[Click here to view code image]

```
class AbstractAnimal {
protected:
    AbstractAnimal& operator=(const AbstractAnimal& rhs);

  public:
    virtual ~AbstractAnimal() = 0;              // see below
    ...

  };
  class Animal: public AbstractAnimal {
  public:
    Animal& operator=(const Animal& rhs);
    ...
  };

  class Lizard: public AbstractAnimal {
  public:
    Lizard& operator=(const Lizard& rhs);
    ...
  };

  class Chicken: public AbstractAnimal {
  public:
    Chicken& operator=(const Chicken& rhs);
    ...
  };
```

This design gives you everything you need. Homogeneous assignments are allowed for lizards, chickens, and animals; partial assignments and heterogeneous assignments are prohibited; and derived class assignment operators may call the assignment operator in the base class. Furthermore, none of the code written in terms of the `Animal`, `Lizard`, or `Chicken` classes requires modification, because these classes continue to exist and to behave as they did before `AbstractAnimal` was introduced. Sure, such code has to be recompiled, but that's a small price to pay for the security of knowing that assignments that compile will behave intuitively and assignments that would behave unintuitively won't compile.

For all this to work, `AbstractAnimal` must be abstract — it must contain at least one pure virtual function. In most cases, coming up with a suitable function is not a problem, but on rare occasions you may find yourself facing the need to create a class like `AbstractAnimal` in which none of the member functions would naturally be declared pure virtual. In such cases, the conventional technique is to make the destructor a pure virtual function; that's what's shown above. In order to support polymorphism through pointers correctly, base classes need virtual destructors anyway, so the only cost associated with making such destructors pure virtual is the inconvenience of having to implement them outside their class definitions. (For an example, see page 195B.)

(If the notion of implementing a pure virtual function strikes you as odd, you just haven't been getting out enough. Declaring a function pure virtual doesn't mean it has no implementation, it means

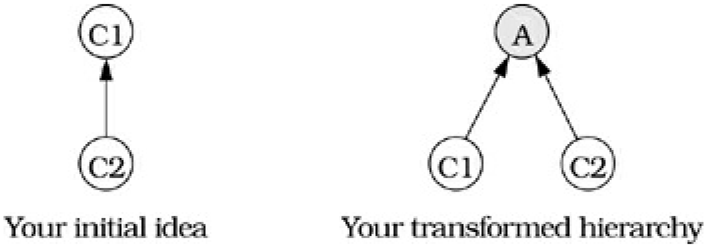
- the current class is abstract, and
- any concrete class inheriting from the current class must declare the function as a "normal" virtual function (i.e., without the "`=0`").

True, most pure virtual functions are never implemented, but pure virtual destructors are a special case. They *must* be implemented, because they are called whenever a derived class destructor is invoked. Furthermore, they often perform useful tasks, such as releasing resources (see Item 9) or logging messages. Implementing pure virtual functions may be uncommon in general, but for pure virtual destructors, it's not just common, it's mandatory.)

You may have noticed that this discussion of assignment through base class pointers is based on the assumption that concrete derived classes like `Lizard` contain data members. If there are no data members in a derived class, you might point out, there is no problem, and it would be safe to have a dataless concrete class inherit from another concrete class. However, just because a class has no data now is no reason to conclude that it will have no data in the future. If it might have data members in the future, all you're doing is postponing the problem until the data members are added, in which case you're merely trading short-term convenience for long-term grief (see also Item 32).

Replacement of a concrete base class like `Animal` with an abstract base class like `AbstractAnimal` yields benefits far beyond simply making the behavior of `operator=` easier to understand. It also reduces the chances that you'll try to treat arrays polymorphically, the unpleasant consequences of which are examined in Item 3. The most significant benefit of the technique, however, occurs at the design level, because replacing concrete base classes with abstract base classes forces you to explicitly recognize the existence of useful abstractions. That is, it makes you create new abstract classes for useful concepts, even if you aren't aware of the fact that the useful concepts exist.

If you have two concrete classes C1 and C2 and you'd like C2 to publicly inherit from C1, you should transform that two-class hierarchy into a three-class hierarchy by creating a new abstract class A and having both C1 and C2 publicly inherit from it:



Your initial idea Your transformed hierarchy

The primary value of this transformation is that it forces you to identify the abstract class A. Clearly, C1 and C2 have something in common; that's why they're related by public inheritance. With this transformation, you must identify what that something is. Furthermore, you must formalize the something as a class in C++, at which point it becomes more than just a vague something, it achieves the status of a formal *abstraction*, one with well-defined member functions and well-defined semantics.

All of which leads to some worrisome thinking. After all, every class represents *some* kind of abstraction, so shouldn't we create two classes for every concept in our hierarchy, one being abstract (to embody the abstract part of the abstraction) and one being concrete (to embody the object-generation part of the abstraction)? No. If you do, you'll end up with a hierarchy with too many classes. Such a hierarchy is difficult to understand, hard to maintain, and expensive to compile. That is not the goal of object-oriented design.

The goal is to identify *useful* abstractions and to force them — and only them — into existence as abstract classes. But how do you identify useful abstractions? Who knows what abstractions might prove useful in the future? Who can predict who's going to want to inherit from what?

Well, I don't know how to predict the future uses of an inheritance hierarchy, but I do know one thing: the need for an abstraction in one context may be coincidental, but the need for an abstraction in more than one context is usually meaningful. Useful abstractions, then, are those that are needed in more than one context. That is, they correspond to classes that are useful in their own right (i.e., it is useful to have objects of that type) and that are also useful for purposes of one or more derived classes.

This is precisely why the transformation from concrete base class to abstract base class is useful: it forces the introduction of a new abstract class only when an existing concrete class is about to be used as a base class, i.e., when the class is about to be (re)used in a new context. Such abstractions are *useful*, because they have, through demonstrated need, shown themselves to be so.

The first time a concept is needed, we can't justify the creation of both an abstract class (for the concept) and a concrete class (for the objects corresponding to that concept), but the second time that concept is needed, we *can* justify the creation of both the abstract and the concrete classes. The transformation I've described simply mechanizes this process, and in so doing it forces designers and programmers to represent explicitly those abstractions that are useful, even if the designers and programmers are not consciously aware of the useful concepts. It also happens to make it a lot easier to bring sanity to the behavior of assignment operators.

Let's consider a brief example. Suppose you're working on an application that deals with moving information between computers on a network by breaking it into packets and transmitting them according to some protocol. All we'll consider here is the class or classes for representing packets. We'll assume such classes make sense for this application.

Suppose you deal with only a single kind of transfer protocol and only a single kind of packet. Perhaps you've heard that other protocols and packet types exist, but you've never supported them, nor do you have any plans to support them in the future. Should you make an abstract class for packets (for the concept that a packet represents) as well as a concrete class for the packets you'll actually be using? If you do, you could hope to add new packet types later without changing the base class for packets. That would save you from having to recompile packet-using applications if you add new packet types. But that design requires two classes, and right now you need only one (for the particular type of packets you use). Is it worth complicating your design now to allow for future extension that may never take place?

There is no unequivocally correct choice to be made here, but experience has shown it is nearly impossible to design good classes for concepts we do not understand well. If you create an abstract class for packets, how likely are you to get it right, especially since your experience is limited to only a single packet type? Remember that you gain the benefit of an abstract class for packets only if you can design that class so that future classes can inherit from it without its being changed in any way. (If it needs to be changed, you have to recompile all packet clients, and you've gained nothing.)

It is unlikely you could design a satisfactory abstract packet class unless you were well versed in many different kinds of packets and in the varied contexts in which they are used. Given your limited experience in this case, my advice would be not to define an abstract class for packets, adding one later only if you find a need to inherit from the concrete packet class.

The transformation I've described here is *a* way to identify the need for abstract classes, not *the* way. There are many other ways to identify good candidates for abstract classes; books on object-oriented analysis are filled with them. It's not the case that the only time you should introduce abstract classes is when you find yourself wanting to have a concrete class inherit from another concrete class. However, the desire to relate two concrete classes by public inheritance is usually indicative of a need for a new abstract class.

As is often the case in such matters, brash reality sometimes intrudes on the peaceful ruminations of theory. Third-party C++ class libraries are proliferating with gusto, and what are you to do if you find yourself wanting to create a concrete class that inherits from a concrete class in a library to which you have only read access?

You can't modify the library to insert a new abstract class, so your choices are both limited and unappealing:

- Derive your concrete class from the existing concrete class, and put up with the assignment-related problems we examined at the beginning of this Item. You'll also have to watch out for the array-related pitfalls described in Item 3.
- Try to find an abstract class higher in the library hierarchy that does most of what you need, then inherit from that class. Of course, there

may not be a suitable class, and even if there is, you may have to duplicate a lot of effort that has already been put into the implementation of the concrete class whose functionality you'd like to extend.

- Implement your new class in terms of the library class you'd like to inherit from. For example, you could have an object of the library class as a data member, then reimplement the library class's interface in your new class:

  Click here to view code image

```
class Window {                 // this is the library class
public:
  virtual void resize(int newWidth, int newHeight);
  virtual void repaint() const;

  int width() const;
  int height() const;
};

class SpecialWindow {          // this is the class you
public:                        // wanted to have inherit
  ...                          // from Window

  // pass-through implementations of nonvirtual
functions
  int width() const { return w.width(); }
  int height() const { return w.height(); }

  // new implementations of "inherited" virtual
functions
  virtual void resize(int newWidth, int newHeight);
  virtual void repaint() const;

private:
  Window w;
};
```

  This strategy requires that you be prepared to update your class each time the library vendor updates the class on which you're dependent. It also requires that you be willing to forgo the ability to redefine virtual functions declared in the library class, because you can't redefine virtual functions unless you inherit them.

- Make do with what you've got. Use the concrete class that's in the

> library and modify your software so that the class suffices. Write non-member functions to provide the functionality you'd like to add to the class, but can't. The resulting software may not be as clear, as efficient, as maintainable, or as extensible as you'd like, but at least it will get the job done.

None of these choices is particularly attractive, so you have to apply some engineering judgment and choose the poison you find least unappealing. It's not much fun, but life's like that sometimes. To make things easier for yourself (and the rest of us) in the future, complain to the vendors of libraries whose designs you find wanting. With luck (and a lot of comments from clients), those designs will improve as time goes on.

Still, the general rule remains: non-leaf classes should be abstract. You may need to bend the rule when working with outside libraries, but in code over which you have control, adherence to it will yield dividends in the form of increased reliability, robustness, comprehensibility, and extensibility throughout your software.

Item 34: Understand how to combine C++ and C in the same program

In many ways, the things you have to worry about when making a program out of some components in C++ and some in C are the same as those you have to worry about when cobbling together a C program out of object files produced by more than one C compiler. There is no way to combine such files unless the different compilers agree on implementation-dependent features like the size of `int`s and `double`s, the mechanism by which parameters are passed from caller to callee, and whether the caller or the callee orchestrates the passing. These pragmatic aspects of mixed-compiler software development are quite properly ignored by language standardization efforts, so the only reliable way to know that object files from compiler A and compiler B can be safely combined in a program is to obtain assurances from the vendors of A and B that their products produce compatible output. This is as true for programs made up of C++ and C as it is for all-C++ or all-C programs, so before you try to mix C++ and C in the same program, make sure your C++ and C compilers generate compatible object files. Having done that, there are four other things you need to consider: name mangling, initialization of statics, dynamic memory allocation, and data structure compatibility.

Name Mangling

Name mangling, as you may know, is the process through which your C++ compilers give each function in your program a unique name. In C, this process is unnecessary, because you can't overload function names, but nearly all C++ programs have at least a few functions with the same name. (Consider, for example, the iostream library, which declares several versions of `operator<<` and `operator>>`.) Overloading is incompatible with most linkers, because linkers generally take a dim view of multiple functions with the same name. Name mangling is a concession to the realities of linkers; in particular, to the fact that linkers usually insist on all function names being unique.

As long as you stay within the confines of C++, name mangling is not likely to concern you. If you have a function name `drawLine` that a compiler mangles into `xyzzy`, you'll always use the name `drawLine`, and you'll have little reason to care that the underlying object files happen to refer to `xyzzy`.

It's a different story if `drawLine` is in a C library. In that case, your C++ source file probably includes a header file that contains a declaration like this,

```
void drawLine(int x1, int y1, int x2, int y2);
```

and your code contains calls to `drawLine` in the usual fashion. Each such call is translated by your compilers into a call to the mangled name of that function, so when you write this,

```
drawLine(a, b, c, d);    // call to unmangled function name
```

your object files contain a function call that corresponds to this:

```
xyzzy(a, b, c, d);       // call to mangled function mame
```

But if `drawLine` is a C function, the object file (or archive or dynamically linked library, etc.) that contains the compiled version of `drawLine` contains a function called *`drawLine`*; no name mangling has taken place. When you try to link the object files comprising your program together, you'll get an error, because the linker is looking for a function called `xyzzy`, and there is no such function.

To solve this problem, you need a way to tell your C++ compilers not to mangle certain function names. You never want to mangle the names of functions written in other languages, whether they be in C, assembler, FORTRAN, Lisp, Forth, or what-have-you. (Yes, what-have-you would include COBOL, but then what would you have?) After all, if you call a C function named `drawLine`, it's really called `drawLine`, and your object code should contain a reference to that name, not to some mangled version of that name.

To suppress name mangling, use C++'s `extern "C"` directive:

[Click here to view code image]

```
// declare a function called drawLine; don't mangle
// its name
extern "C"
void drawLine(int x1, int y1, int x2, int y2);
```

Don't be drawn into the trap of assuming that where there's an `extern "C"`, there must be an `extern "Pascal"` and an `extern "FORTRAN"` as well. There's not, at least not in the standard. The best way to view `extern "C"` is not as an assertion that the associated function is written in C, but as a statement that the function should be called as if it *were* written in C. (Technically, `extern "C"` means the function has C linkage, but what that means is far from clear. One thing it always means, however, is that name mangling is suppressed.)

For example, if you were so unfortunate as to have to write a function in assembler, you could declare it `extern "C"`, too:

[Click here to view code image]

```
// this function is in assembler - don't mangle its name
extern "C" void twiddleBits(unsigned char bits);
```

You can even declare C++ functions `extern "C"`. This can be useful if you're writing a library in C++ that you'd like to provide to clients using other programming languages. By suppressing the name mangling of your C++ function names, your clients can use the natural and intuitive names you choose instead of the mangled names your compilers would otherwise generate:

[Click here to view code image]

```
// the following C++ function is designed for use outside
// C++ and should not have its name mangled
extern "C" void simulate(int iterations);
```

Often you'll have a slew of functions whose names you don't want mangled, and it would be a pain to precede each with `extern "C"`. Fortunately, you don't have to. `extern "C"` can also be made to apply to a whole set of functions. Just enclose them all in curly braces:

[Click here to view code image]

```
extern "C" {                  // disable name mangling for
                              // all the following functions

  void drawLine(int x1, int y1, int x2, int y2);
  void twiddleBits(unsigned char bits);
  void simulate(int iterations);
  ...

}
```

This use of `extern "C"` simplifies the maintenance of header files that must be used with both C++ and C. When compiling for C++, you'll want to include `extern "C"`, but when compiling for C, you won't. By taking advantage of the fact that the preprocessor symbol `__cplusplus` is defined only for C++ compilations, you can structure your polyglot header files as follows:

[Click here to view code image]

```
#ifdef __cplusplus

extern "C" {

#endif

  void drawLine(int x1, int y1, int x2, int y2);
  void twiddleBits(unsigned char bits);
  void simulate(int iterations);
  ...

#ifdef __cplusplus

}

#endif
```

There is, by the way, no such thing as a "standard" name mangling algorithm.

Different compilers are free to mangle names in different ways, and different compilers do. This is a good thing. If all compilers mangled names the same way, you might be lulled into thinking they all generated compatible code. The way things are now, if you try to mix object code from incompatible C++ compilers, there's a good chance you'll get an error during linking, because the mangled names won't match up. This implies you'll probably have other compatibility problems, too, and it's better to find out about such incompatibilities sooner than later,

Initialization of Statics

Once you've mastered name mangling, you need to deal with the fact that in C++, lots of code can get executed before and after `main`. In particular, the constructors of static class objects and objects at global, namespace, and file scope are usually called before the body of `main` is executed. This process is known as *static initialization*. This is in direct opposition to the way we normally think about C++ and C programs, in which we view `main` as the entry point to execution of the program. Similarly, objects that are created through static initialization must have their destructors called during *static destruction*; that process typically takes place after `main` has finished executing.

To resolve the dilemma that `main` is supposed to be invoked first, yet objects need to be constructed before `main` is executed, many compilers insert a call to a special compiler-written function at the beginning of `main`, and it is this special function that takes care of static initialization. Similarly, compilers often insert a call to another special function at the end of `main` to take care of the destruction of static objects. Code generated for `main` often looks as if `main` had been written like this:

Click here to view code image

```
int main(int argc, char *argv[])
{
  performStaticInitialization();     // generated by the
                                     // implementation

  the statements you put in main go here;

  performStaticDestruction();        // generated by the
                                     // implementation
}
```

Now don't take this too literally. The functions `performStaticInitialization` and `performStaticDestruction` usually have much more cryptic names, and they may even be generated inline, in which case you won't see any functions for them in your object files. The important point is this: if a C++ compiler adopts this approach to the initialization and destruction of static objects, such objects will be neither initialized nor destroyed unless `main` is written in C++. Because this approach to static initialization and destruction is common, you should try to write `main` in C++ if you write any part of a software system in C++.

Sometimes it would seem to make more sense to write `main` in C — say if most of a program is in C and C++ is just a support library. Nevertheless, there's a good chance the C++ library contains static objects (if it doesn't now, it probably will in the future — see Item 32), so it's still a good idea to write `main` in C++ if you possibly can. That doesn't mean you need to rewrite your C code, however. Just rename the `main` you wrote in C to be `realMain`, then have the C++ version of `main` call `realMain`:

Click here to view code image

```
extern "C"                              // implement this
int realMain(int argc, char *argv[]);   // function in C

int main(int argc, char *argv[])        // write this in C++
{
  return realMain(argc, argv);
}
```

If you do this, it's a good idea to put a comment above `main` explaining what is going on.

If you cannot write `main` in C++, you've got a problem, because there is no other portable way to ensure that constructors and destructors for static objects are called. This doesn't mean all is lost, it just means you'll have to work a little harder. Compiler vendors are well acquainted with this problem, so almost all provide some extralinguistic mechanism for initiating the process of static initialization and static destruction. For information on how this works with your compilers, dig into your compilers' documentation or contact their vendors.

Dynamic Memory Allocation

That brings us to dynamic memory allocation. The general rule is simple: the C++ parts of a program use `new` and `delete` (see Item 8), and the C parts of a program use `malloc` (and its variants) and `free`. As long as memory that came from `new` is deallocated via `delete` and memory that came from `malloc` is deallocated via `free`, all is well. Calling `free` on a `new`ed pointer yields undefined behavior, however, as does `delete`ing a `malloc`ed pointer. The only thing to remember, then, is to segregate rigorously your `new`s and `delete`s from your `malloc`s and `free`s.

Sometimes this is easier said than done. Consider the humble (but handy) `strdup` function, which, though standard in neither C nor C++, is nevertheless widely available:

Click here to view code image

```
char * strdup(const char *ps);    // return a copy of the
                                  // string pointed to by ps
```

If a memory leak is to be avoided, the memory allocated inside `strdup` must be deallocated by `strdup`'s caller. But how is the memory to be deallocated? By using `delete`? By calling `free`? If the `strdup` you're calling is from a C library, it's the latter. If it was written for a C++ library, it's probably the former. What you need to do after calling `strdup`, then, varies not only from system to system, but also from compiler to compiler. To reduce such portability headaches, try to avoid calling functions that are neither in the standard library (see Item 35) nor available in a stable form on most computing platforms.

Data Structure Compatibility

Which brings us at long last to passing data between C++ and C programs. There's no hope of making C functions understand C++ features, so the level of discourse between the two languages must be limited to those concepts that C can express. Thus, it should be clear there's no portable way to pass objects or to pass pointers to member functions to routines written in C. C does understand normal pointers, however, so, provided your C++ and C compilers produce compatible output, functions in the two languages can safely exchange pointers to objects and pointers to non-member or static functions. Naturally, structs and variables of built-in types (e.g., `int`s, `char`s, etc.) can also freely cross the C++/C border.

Because the rules governing the layout of a `struct` in C++ are consistent with those of C, it is safe to assume that a structure definition that compiles in both languages is laid out the same way by both compilers. Such structs can be safely passed back and forth between C++ and C. If you add *nonvirtual* functions to the C++ version of the struct, its memory layout should not change, so objects of a struct (or class) containing only non-virtual functions should be compatible with their C brethren whose structure definition lacks only the member function declarations. Adding *virtual* functions ends the game, because the addition of virtual functions to a class causes objects of that type to use a different memory layout (see Item 24). Having a struct inherit from another struct (or class) usually changes its layout, too, so structs with base structs (or classes) are also poor candidates for exchange with C functions.

From a data structure perspective, it boils down to this: it is safe to pass data structures from C++ to C and from C to C++ provided the definition of those structures compiles in both C++ and C. Adding nonvirtual member functions to the C++ version of a struct that's otherwise compatible with C will probably not affect its compatibility, but almost any other change to the struct will.

Summary

If you want to mix C++ and C in the same program, remember the following simple guidelines:

- Make sure the C++ and C compilers produce compatible object files.
- Declare functions to be used by both languages `extern "C"`.
- If at all possible, write `main` in C++.
- Always use `delete` with memory from `new`; always use `free` with memory from `malloc`.
- Limit what you pass between the two languages to data structures that compile under C; the C++ version of structs may contain nonvirtual member functions.

Item 35: Familiarize yourself with the language standard

Since its publication in 1990, *The Annotated C++ Reference Manual* (see page 285B) has been the definitive reference for working programmers needing to know what is in C++ and what is not. In the years since the ARM (as it's fondly known) came out, the ISO/ANSI committee standardizing the language has changed (primarily extended) the language in ways both big and small. As a definitive reference, the ARM no longer suffices.

The post-ARM changes to C++ significantly affect how good programs are written. As a result, it is important for C++ programmers to be familiar with the primary ways in which the C++ specified by the standard differs from that described by the ARM.

The ISO/ANSI standard for C++ is what vendors will consult when implementing compilers, what authors will examine when preparing books, and what programmers will look to for definitive answers to questions about C++. Among the most important changes to C++ since the ARM are the following:

- **New features have been added:** RTTI, namespaces, `bool`, the `mutable` and `explicit` keywords, the ability to overload operators for enums, and the ability to initialize constant integral static class members within a class definition.
- **Templates have been extended:** member templates are now allowed, there is a standard syntax for forcing template instantiations, non-type arguments are now allowed in function templates, and class templates may themselves be used as template arguments.
- **Exception handling has been refined:** exception specifications are now more rigorously checked during compilation, and the `unexpected` function may now throw a `bad_exception` object.
- **Memory allocation routines have been modified:** `operator new[]` and `operator delete[]` have been added, the operators `new`/`new[]` now throw an exception if memory can't be allocated, and there are now alternative versions of the operators `new`/`new[]` that return 0 when an allocation fails.
- **New casting forms have been added:** `static_cast`, `dynamic_cast`, `const_cast`, and `reinterpret_cast`.
- **Language rules have been refined:** redefinitions of virtual functions need no longer have a return type that exactly matches that of the function they redefine, and the lifetime of temporary objects has been

defined precisely.

Almost all these changes are described in *The Design and Evolution of C++* (see page 285B). Current C++ textbooks (those written after 1994) should include them, too. (If you find one that doesn't, reject it.) In addition, *More Effective C++* (that's this book) contains examples of how to use most of these new features. If you're curious about something on this list, try looking it up in the index.

The changes to C++ proper pale in comparison to what's happened to the standard library. Furthermore, the evolution of the standard library has not been as well publicized as that of the language. *The Design and Evolution of C++*, for example, makes almost no mention of the standard library. The books that do discuss the library are sometimes out of date, because the library changed quite substantially in 1994.

The capabilities of the standard library can be broken down into the following general categories:

- **Support for the standard C library.** Fear not, C++ still remembers its roots. Some minor tweaks have brought the C++ version of the C library into conformance with C++'s stricter type checking, but for all intents and purposes, everything you know and love (or hate) about the C library continues to be knowable and lovable (or hateable) in C++, too.
- **Support for strings.** As Chair of the working group for the standard C++ library, Mike Vilot was told, "If there isn't a standard `string` type, there will be blood in the streets!" (Some people get so emotional.) Calm yourself and put away those hatchets and truncheons — the standard C++ library has strings.
- **Support for localization.** Different cultures use different character sets and follow different conventions when displaying dates and times, sorting strings, printing monetary values, etc. Localization support within the standard library facilitates the development of programs that accommodate such cultural differences.
- **Support for I/O.** The iostream library remains part of the C++ standard, but the committee has tinkered with it a bit. Though some classes have been eliminated (notably `iostream` and `fstream`) and some have been replaced (e.g., `string`-based `stringstream`s replace `char*`-based `strstream`s,

which are now deprecated), the basic capabilities of the standard iostream classes mirror those of the implementations that have existed for several years.

- **Support for numeric applications.** Complex numbers, long a mainstay of examples in C++ texts, have finally been enshrined in the standard library. In addition, the library contains special array classes (`valarray`s) that restrict aliasing. These arrays are eligible for more aggressive optimization than are built-in arrays, especially on multiprocessing architectures. The library also provides a few commonly useful numeric functions, including partial sum and adjacent difference.
- **Support for general-purpose containers and algorithms.** Contained within the standard C++ library is a set of class and function templates collectively known as the Standard Template Library (STL). The STL is the most revolutionary part of the standard C++ library. I summarize its features below.

Before I describe the STL, though, I must dispense with two idiosyncrasies of the standard C++ library you need to know about.

First, almost everything in the library is a *template*. In this book, I may have referred to the standard `string` class, but in fact there is no such class. Instead, there is a class template called `basic_string` that represents sequences of characters, and this template takes as a parameter the type of the characters making up the sequences. This allows for strings to be made up of `char`s, wide chars, Unicode chars, whatever.

What we normally think of as the `string` class is really the template instantiation `basic_string<char>`. Because its use is so common, the standard library provides a typedef:

```
typedef basic_string<char> string;
```

Even this glosses over many details, because the `basic_string` template takes three arguments; all but the first have default values. To *really* understand the `string` type, you must face this full, unexpurgated declaration of `basic_string`:

Click here to view code image

```
template<class charT,
         class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
  class basic_string;
```

You don't need to understand this gobbledygook to use the `string` type, because even though `string` is a typedef for The Template Instantiation from Hell, it behaves as if it were the unassuming non-template class the typedef makes it appear to be. Just tuck away in the back of your mind the fact that if you ever need to customize the types of characters that go into strings, or if you want to fine-tune the behavior of those characters, or if you want to seize control over the way memory for strings is allocated, the `basic_string` template allows you to do these things.

The approach taken in the design of the `string` type — generalize it and make the generalization a template — is repeated throughout the standard C++ library. IOstreams? They're templates; a type parameter defines the type of character making up the streams. Complex numbers? Also templates; a type parameter defines how the components of the numbers should be stored. Valarrays? Templates; a type parameter specifies what's in each array. And of course the STL consists almost entirely of templates. If you are not comfortable with templates, now would be an excellent time to start making serious headway toward that goal.

The other thing to know about the standard library is that virtually everything it contains is inside the namespace `std`. To use things in the standard library without explicitly qualifying their names, you'll have to employ a `using` directive or (preferably) `using` declarations. Fortunately, this syntactic administrivia is automatically taken care of when you `#include` the appropriate headers.

The Standard Template Library

The biggest news in the standard C++ library is the STL, the Standard Template Library. (Since almost everything in the C++ library is a template, the name STL is not particularly descriptive. Nevertheless, this is the name of the containers and algorithms portion of the library, so good name or bad, this is what we use.)

The STL is likely to influence the organization of many — perhaps most — C++ libraries, so it's important that you be familiar with its general principles. They are not difficult to understand. The STL is based on three fundamental concepts: containers, iterators, and algorithms. Containers hold collections of objects. Iterators are pointer-like objects that let you walk through STL containers just as you'd use pointers to walk through built-in arrays. Algorithms are functions that work on STL containers and that use iterators to help them do their work.

It is easiest to understand the STL view of the world if we remind ourselves of the C++ (and C) rules for arrays. There is really only one rule we need to know: a pointer to an array can legitimately point to any element of the array *or to one element beyond the end of the array*. If the pointer points to the element beyond the end of the array, it can be compared only to other pointers to the array; the results of dereferencing it are undefined.

We can take advantage of this rule to write a function to find a particular value in an array. For an array of integers, our function might look like this:

Click here to view code image

```
int * find(int *begin, int *end, int value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

This function looks for `value` in the range between `begin` and `end` (excluding `end` – `end` points to one beyond the end of the array) and returns a pointer to the first occurrence of `value` in the array; if none is found, it returns `end`.

Returning `end` seems like a funny way to signal a fruitless search. Wouldn't 0 (the null pointer) be better? Certainly null seems more natural, but that doesn't make it "better." The `find` function must return some distinctive pointer value to indicate the search failed, and for this purpose, the `end` pointer is as good as the null pointer. In addition, as we'll soon see, the `end` pointer generalizes to other types of containers better than the null pointer.

Frankly, this is probably not the way you'd write the `find` function, but it's not unreasonable, and it generalizes astonishingly well. If you followed this simple example, you have mastered most of the ideas on which the STL is founded.

You could use the `find` function like this:

Click here to view code image

```
int values[50];
```

```
...

int *firstFive = find(values,        // search the range
                      values+50,    // values[0] -
values[49]
                      5);           // for the value 5

if (firstFive != values+50) {       // did the search
succeed?

  ...                               // yes

}
else {
  ...                               // no, the search failed
}
```

You can also use `find` to search subranges of the array:

[Click here to view code image]

```
int *firstFive = find(values,        // search the range
                      values+10,    // values[0] - values[9]
                      5);           // for the value 5

int age = 36;

...

int *firstValue = find(values+10,   // search the range
                       values+20,   // values[10] -
values[19]
                       age);        // for the value in age
```

There's nothing inherent in the `find` function that limits its applicability to arrays of `int`s, so it should really be a template:

[Click here to view code image]

```
template<class T>
T * find(T *begin, T *end, const T& value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

In the transformation to a template, notice how we switched from pass-by-value for `value` to pass-by-reference-to-`const`. That's because now that we're passing arbitrary types around, we have to worry about the cost of pass-by-value. Each by-value parameter costs us a call to the parameter's constructor and destructor every time the function is invoked. We avoid these costs by using pass-by-reference, which involves no object construction or destruction.

This template is nice, but it can be generalized further. Look at the operations on `begin` and `end`. The only ones used are comparison for inequality, dereferencing, prefix increment (see Item 6), and copying (for the function's return value — see Item 19). These are all operations we can overload, so why limit `find` to using pointers? Why not allow *any* object that supports these operations to be used in addition to pointers? Doing so would free the `find` function from the built-in meaning of pointer operations. For example, we could define a pointer-like object for a linked list whose prefix increment operator moved us to the next element in the list.

This is the concept behind STL *iterators*. Iterators are pointer-like objects designed for use with STL containers. They are first cousins to the smart pointers of Item 28, but smart pointers tend to be more ambitious in what they do than do STL iterators. From a technical viewpoint, however, they are implemented using the same techniques.

Embracing the notion of iterators as pointer-like objects, we can replace the pointers in `find` with iterators, thus rewriting `find` like this:

Click here to view code image

```
template<class Iterator, class T>
Iterator find(Iterator begin, Iterator end, const T& value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

Congratulations! You have just written part of the Standard Template Library. The STL contains dozens of algorithms that work with containers and iterators, and `find` is one of them.

Containers in STL include `bitset`, `vector`, `list`, `deque`, `queue`, `priority_queue`, `stack`, `set`, and `map`, and you can apply `find` to *any* of these container types:

Click here to view code image

```
list<char> charList;           // create STL list object
                               // for holding chars

...

// find the first occurrence of 'x' in charList
list<char>::iterator it = find(charList.begin(),
                               charList.end(),
                               'x');
```

"Whoa!", I hear you cry, "This doesn't look *anything* like it did in the array examples above!" Ah, but it does; you just have to know what to look for.

To call `find` for a `list` object, you need to come up with iterators that point to the first element of the list and to one past the last element of the list. Without some help from the `list` class, this is a difficult task, because you have no idea how a `list` is implemented. Fortunately, `list` (like all STL containers) obliges by providing the member functions `begin` and `end`. These member functions return the iterators you need, and it is those iterators that are passed into the first two parameters of `find` above.

When `find` is finished, it returns an iterator object that points to the found element (if there is one) or to `charList.end()` (if there's not). Because you know nothing about how `list` is implemented, you also know nothing about how iterators into `list`s are implemented. How, then, are you to know what type of object is returned by `find`? Again, the `list` class, like all STL containers, comes to the rescue: it provides a typedef, `iterator`, that is the type of iterators into `list`s. Since `charList` is a `list` of `char`s, the type of an iterator into such a list is `list<char>::iterator`, and that's what's used in the example above. (Each STL container class actually defines two iterator types, `iterator` and `const_iterator`. The former acts like a normal pointer, the latter like a pointer-to-`const`.)

Exactly the same approach can be used with the other STL containers. Furthermore, C++ pointers *are* STL iterators, so the original array examples work with the STL `find` function, too:

Click here to view code image

```
int values[50];

...
```

```
int *firstFive = find(values, values+50, 5);  // fine,
calls
                                              // STL find
```

At its core, STL is very simple. It is just a collection of class and function templates that adhere to a set of conventions. The STL collection classes provide functions like `begin` and `end` that return iterator objects of types defined by the classes. The STL algorithm functions move through collections of objects by using iterator objects over STL collections. STL iterators act like pointers. That's really all there is to it. There's no big inheritance hierarchy, no virtual functions, none of that stuff. Just some class and function templates and a set of conventions to which they all subscribe.

Which leads to another revelation: STL is extensible. You can add your own collections, algorithms, and iterators to the STL family. As long as you follow the STL conventions, the standard STL collections will work with your algorithms and your collections will work with the standard STL algorithms. Of course, your templates won't be part of the standard C++ library, but they'll be built on the same principles and will be just as reusable.

There is much more to the C++ library than I've described here. Before you can use the library effectively, you must learn more about it than I've had room to summarize, and before you can write your own STL-compliant templates, you must learn more about the conventions of the STL. The standard C++ library is far richer than the C library, and the time you take to familiarize yourself with it is time well spent. Furthermore, the design principles embodied by the library — those of generality, extensibility, customizability, efficiency, and reusability — are well worth learning in their own right. By studying the standard C++ library, you not only increase your knowledge of the ready-made components available for use in your software, you learn how to apply the features of C++ more effectively, and you gain insight into how to design better libraries of your own.

Recommended Reading

So your appetite for information on C++ remains unsated. Fear not, there's more — much more. In the sections that follow, I put forth my recommendations for further reading on C++. It goes without saying that such recommendations are both subjective and selective, but in view of the litigious age in which we live, it's probably a good idea to say it anyway.

Books

There are hundreds — possibly thousands — of books on C++, and new contenders join the fray with great frequency. I haven't seen all these books, much less read them, but my experience has been that while some books are very good, some of them, well, some of them aren't.

What follows is the list of books I find myself consulting when I have questions about software development in C++. Other good books are available, I'm sure, but these are the ones I use, the ones I can truly *recommend*.

A good place to begin is with the books that describe the language itself. Unless you are crucially dependent on the nuances of the official standards documents, I suggest you do, too.

The Annotated C++ Reference Manual, Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley, 1990, ISBN 0-201-51459-1.

The Design and Evolution of C++, Bjarne Stroustrup, Addison-Wesley, 1994, ISBN 0-201-54330-3.

These books contain not just a description of what's in the language, they also explain the rationale behind the design decisions — something you won't find in the official standard documents. *The Annotated C++ Reference Manual* is now incomplete (several language features have been added since it was published — see Item 35) and is in some cases out of date, but it is still the best reference for the core parts of the language, including templates and exceptions. *The Design and Evolution of C++* covers most of what's missing in *The Annotated C++ Reference Manual*; the only thing it lacks is a discussion of the Standard Template Library (again, see Item 35). These books are not tutorials, they're references, but you can't truly understand C++ unless you understand the material in these books.

For a more general reference on the language, the standard library, and how to apply it, there is no better place to look than the book by the man responsible for C++ in the first place:

The C++ Programming Language (Third Edition), Bjarne Stroustrup, Addison-Wesley, 1997, ISBN 0-201-88954-4.

Stroustrup has been intimately involved in the language's design, implementation, application, and standardization since its inception, and he probably knows more about it than anybody else does. His descriptions of language features make for dense reading, but that's primarily because they contain so much information. The chapters on the standard C++ library provide a good introduction to this crucial aspect of modern C++.

If you're ready to move beyond the language itself and are interested in how to apply it effectively, you might consider my first book on the subject:

Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs, Scott Meyers, Addison-Wesley, 2005, ISBN 0-321-33487-6.

That book is organized similarly to this one, but it covers different (arguably more fundamental) material.

A book pitched at roughly the same level as my *Effective C++* books, but covering different topics, is

C++ Strategies and Tactics, Robert Murray, Addison-Wesley, 1993, ISBN 0-201-56382-7.

Murray's book is especially strong on the fundamentals of template design, a topic to which he devotes two chapters. He also includes a chapter on the important topic of migrating from C development to C++ development. Much of my discussion on reference counting (see Item 29) is based on the ideas in *C++ Strategies and Tactics*.

If you're the kind of person who likes to learn proper programming technique by reading *code*, the book for you is

C++ Programming Style, Tom Cargill, Addison-Wesley, 1992, ISBN 0-201-56365-7.

Each chapter in this book starts with some C++ software that has been published as an example of how to do something correctly. Cargill then proceeds to dissect — nay, *vivisect* — each program, identifying likely trouble spots, poor design choices, brittle implementation decisions, and things that are just plain wrong. He then iteratively rewrites each example to eliminate the weaknesses, and by the time he's done, he's produced code that is more robust, more maintainable, more efficient, and more portable, and it still fulfills the original problem specification. Anybody programming in C++ would do well to heed the lessons of this book, but it is especially important for those involved in code inspections.

One topic Cargill does not discuss in *C++ Programming Style* is exceptions. He turns his critical eye to this language feature in the following article, however, which demonstrates why writing exception-safe code is more difficult than most programmers realize:

"Exception Handling: A False Sense of Security," *C++ Report*, Volume 6, Number 9, November-December 1994, pages 21-24.

If you are contemplating the use of exceptions, read this article before you proceed. If you don't have access to back issues of the *C++ Report*, you can find the article at the Addison-Wesley Internet site. The World Wide Web URL is `http://www.awl.com/cp/mec++.html`. If you prefer anonymous FTP, you can get the article from `ftp.awl.com` in the directory cp/mec++.

Once you've mastered the basics of C++ and are ready to start pushing the envelope, you must familiarize yourself with

Advanced C++: Programming Styles and Idioms, James Coplien, Addison-Wesley, 1992, ISBN 0-201-54855-0.

I generally refer to this as "the LSD book," because it's purple and it will expand your mind. Coplien covers some straightforward material, but his focus is really on showing you how to do things in C++ you're not supposed to be able to do. You want to construct objects on top of one another? He shows you how. You want to bypass strong typing? He gives you a way. You want to add data and functions to classes as your programs are running? He explains how to do it. Most of the time, you'll want to steer clear of the techniques he describes, but sometimes they provide just the solution you need for a tricky problem you're facing. Furthermore, it's illuminating just to see what kinds of things can be done with C++. This book may frighten you, it may dazzle you, but when you've read it, you'll never look at C++ the same way again.

If you have anything to do with the design and implementation of C++ libraries, you would be foolhardy to overlook

Designing and Coding Reusable C++, Martin D. Carroll and Margaret A. Ellis, Addison-Wesley, 1995, ISBN 0-201-51284-X.

Carroll and Ellis discuss many practical aspects of library design and implementation that are simply ignored by everybody else. Good libraries are small, fast, extensible, easily upgraded, graceful during template instantiation, powerful, and robust. It is not possible to optimize for each of these attributes, so one must make trade-offs that improve some aspects of a library at the expense of others. *Designing and Coding Reusable C++* examines these trade-offs and offers down-to-earth advice on how to go about making them.

Regardless of whether you write software for scientific and engineering applications, you owe yourself a look at

Scientific and Engineering C++, John J. Barton and Lee R. Nackman, Addison-Wesley, 1994, ISBN 0-201-53393-6.

The first part of the book explains C++ for FORTRAN programmers (now *there's* an unenviable task), but the latter parts cover techniques that are relevant in virtually any domain. The extensive material on templates is close to revolutionary; it's probably the most advanced that's currently available, and I suspect that when you've seen the miracles these authors perform with templates, you'll never again think of them as little more than souped-up macros.

Finally, the emerging discipline of *patterns* in object-oriented software development (see page 123B) is described in

Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995, ISBN 0-201-63361-2.

This book provides an overview of the ideas behind patterns, but its primary contribution is a catalogue of 23 fundamental patterns that are useful in many application areas. A stroll through these pages will almost surely reveal a pattern you've had to invent yourself at one time or another, and when you find one, you're almost certain to discover that the design in the book is superior to the ad-hoc approach you came up with. The names of the patterns here have already become part of an emerging vocabulary for object-oriented design; failure to know these names may soon be hazardous to your ability to communicate with your colleagues. A particular strength of the book is its emphasis on designing and implementing software so that future evolution is gracefully accommodated (see Items 32 and 33).

Design Patterns is also available as a CD-ROM:

Design Patterns CD: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1998, ISBN 0-201-63498-8.

Magazines†

For hard-core C++ programmers, there's really only one game in town:

C++ Report, SIGS Publications, New York, NY.

The magazine has made a conscious decision to move away from its "C++ only" roots, but the increased coverage of domain- and system-specific programming issues is worthwhile in its own right, and the material on C++, if occasionally a bit off the deep end, continues to be the best available.

If you're more comfortable with C than with C++, or if you find the *C++ Report*'s material too extreme to be useful, you may find the articles in this magazine more to your taste:

C/C++ Users Journal, Miller Freeman, Inc., Lawrence, KS.

As the name suggests, this covers both C and C++. The articles on C++ tend to assume a weaker background than those in the *C++ Report*. In addition, the editorial staff keeps a tighter rein on its authors than does the *Report*, so the material in the magazine tends to be relatively mainstream. This helps filter out ideas on the lunatic fringe, but it also limits your exposure to techniques that are truly cutting-edge.

Usenet Newsgroups

Three Usenet newsgroups are devoted to C++. The general-purpose anything-goes newsgroup is `comp.lang.c++`. The postings there run the gamut from detailed explanations of advanced programming techniques to rants and raves by those who love or hate C++ to undergraduates the world over asking for help with the homework assignments they neglected until too late. Volume in the newsgroup is extremely high. Unless you have hours of free time on your hands, you'll want to employ a filter to help separate the wheat from the chaff. Get a good filter — there's a lot of chaff.

In November 1995, a moderated version of `comp.lang.c++` was created. Named `comp.lang.c++.moderated`, this newsgroup is also designed for general discussion of C++ and related issues, but the moderators aim to weed out implementation-specific questions and comments, questions covered in the extensive on-line FAQ ("Frequently Asked Questions" list), flame wars, and other matters of little interest to most C++ practitioners.

A more narrowly focused newsgroup is `comp.std.c++`, which is devoted to a discussion of the C++ standard itself. Language lawyers abound in this group, but it's a good place to turn if your picky questions about C++ go unanswered in the references otherwise available to you. The newsgroup is moderated, so the signal-to-noise ratio is quite good; you won't see any pleas for homework assistance here.

An `auto_ptr` Implementation

Items 9, 10, 26, 31 and 32 attest to the remarkable utility of the `auto_ptr` template. Unfortunately, few compilers currently ship with a "correct" implementation.[†] Items 9 and 28 sketch how you might write one yourself, but it's nice to have more than a sketch when embarking on real-world projects.

Below are two presentations of an implementation for `auto_ptr`. The first presentation documents the class interface and implements all the member functions outside the class definition. The second implements each member function within the class definition. Stylistically, the second presentation is inferior to the first, because it fails to separate the class interface from its implementation. However, `auto_ptr` yields simple classes, and the second presentation brings that out much more clearly than does the first.

Here is `auto_ptr` with its interface documented:

Click here to view code image

```
template<class T>
class auto_ptr {
public:
  explicit auto_ptr(T *p = 0);  // see Item 5 for a
                                // description of
"explicit"

  template<class U>             // copy constructor member
  auto_ptr(auto_ptr<U>& rhs);   // template (see 28):
                                // initialize a new
auto_ptr
                                // with any compatible
                                // auto_ptr

  ~auto_ptr();

  template<class U>             // assignment operator
  auto_ptr<T>&                  // member template (see
  operator=(auto_ptr<U>& rhs);  // 28): assign from any
                                // compatible auto_ptr
  T& operator*() const;         // see 28
  T* operator->() const;        // see 28

  T* get() const;               // return value of current
                                // dumb pointer

  T* release();                 // relinquish ownership of
                                // current dumb pointer and
                                // return its value

  void reset(T *p = 0);         // delete owned pointer;
                                // assume ownership of p
private:
  T *pointee;

template<class U>               // make all auto_ptr
classes
friend class auto_ptr<U>;       // friends of one another
};

template<class T>
inline auto_ptr<T>::auto_ptr(T *p)
: pointee(p)
{}

template<class T>
  template<class U>
  inline auto_ptr<T>::auto_ptr(auto_ptr<U>& rhs)
  : pointee(rhs.release())
  {}

template<class T>
inline auto_ptr<T>::~auto_ptr()
{ delete pointee; }

template<class T>
  template<class U>
  inline auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<U>&
rhs)
  {
    if (this != &rhs) reset(rhs.release());
    return *this;
  }

template<class T>
```

```
inline T& auto_ptr<T>::operator*() const
{ return *pointee; }

template<class T>
inline T* auto_ptr<T>::operator->() const
{ return pointee; }

template<class T>
inline T* auto_ptr<T>::get() const
{ return pointee; }

template<class T>
inline T* auto_ptr<T>::release()
{
  T *oldPointee = pointee;
  pointee = 0;
  return oldPointee;
}

template<class T>
inline void auto_ptr<T>::reset(T *p)
{
  if (pointee != p) {
    delete pointee;
    pointee = p;
  }
}
```

Here is `auto_ptr` with all the functions defined in the class definition. As you can see, there's no brain surgery going on here:

Click here to view code image

```
template<class T>
class auto_ptr {
public:
  explicit auto_ptr(T *p = 0): pointee(p) {}

  template<class U>
  auto_ptr(auto_ptr<U>& rhs): pointee(rhs.release()) {}

  ~auto_ptr() { delete pointee; }

  template<class U>
  auto_ptr<T>& operator=(auto_ptr<U>& rhs)
  {
      if (this != &rhs) reset(rhs.release());
      return *this;
    }

    T& operator*() const { return *pointee; }

    T* operator->() const { return pointee; }

    T* get() const { return pointee; }

    T* release()
    {
      T *oldPointee = pointee;
      pointee = 0;
      return oldPointee;
    }

    void reset(T *p = 0)
    {
      if (pointee != p) {
        delete pointee;
        pointee = p;
      }
    }
  private:
    T *pointee;

  template<class U> friend class auto_ptr<U>;
  };
```

If your compilers don't yet support `explicit`, you may safely `#define` it out of existence:

```
  #define explicit
```

This won't make `auto_ptr` any less functional, but it will render it slightly less safe. For details, see Item 5.

If your compilers lack support for member templates, you can use the non-template `auto_ptr` copy constructor and assignment operator described in Item 28. This will make your `auto_ptr`s less convenient to use, but there is, alas, no way to approximate the behavior of member templates. If member templates (or other language features, for that matter) are important to you, let your compiler vendors know. The more customers ask for new language features, the sooner vendors will implement them.

General Index

This index is for everything in this book except the classes, functions, and templates I use as examples. If you're looking for a reference to a particular class, function, or template I use as an example, please turn to the index beginning on page 313. For everything else, this is the place to be. In particular, classes, functions, and templates in the standard C++ library (e.g., `string`, `auto_ptr`, `list`, `vector`, etc.) are indexed here.

For the most part, operators are listed under *operator*. For example, `operator<<` is listed under `operator<<`, not under `<<`, etc. However, operators whose names are words or are word-like (e.g., `new`, `delete`, `sizeof`, `const_cast`, etc.) are listed under the appropriate words (e.g., `new`, `delete`, `sizeof`, `const_cast`, etc.).

Example uses of new and lesser-known language features are indexed under *example uses*.

Before A

`#define`, 294B

`?:`, vs. `if`/`then`, 56B

`__cplusplus`, 273B

“`>>`”, vs. “`> >`”, 29B

80-20 rule, 79B, 82–85B, 106B

90-10 rule, 82B

A

`abort`

- and `assert`, 167B
- and object destruction, 72B
- relationship to `terminate`, 72B

abstract classes

- and inheritance, 258–270B
- and vtbls, 115B
- drawing, 5B
- identifying, 267B, 268B
- transforming from concrete, 266–269B

abstract mixin base classes, 154B

abstractions
- identifying, 267B, 268B
- useful, 267B

access-declarations, 144B

adding data and functions to classes at runtime, 287B

Addison-Wesley Internet site, 8B, 287B

address comparisons to determine object locations, 150–152B

address-of operator — see `operator&`

Advanced C++: Programming Styles and Idioms, 287B

Adventure, allusion to, 271B

allocation of memory — see *memory allocation*

amortizing computational costs, 93–98B

Annotated C++ Reference Manual, The, 277B, 285B

ANSI/ISO standardization committee, 2B, 59B, 96B, 256B, 277B

APL, 92B

application framework, 234B

approximating
- `bool`, 3–4B
- C++-style casts, 15–16B
- `const static` data members, 141B
- `explicit`, 29–31B
- in-class `using` declarations, 144B
- member templates, 294B
- `mutable`, 89–90B

  - virtual functions, 121B
  - vtbls, 235–251B
- ARM, the, 277B
- array `new`, 42B
- arrays
  - and `auto_ptr`, 48B
  - and default constructors, 19–21B
  - and inheritance, 17–18B
  - and pointer arithmetic, 17B
  - associative, 236B, 237B
  - dynamic, 96–98B
  - memory allocation for, 42–43B
  - multi-dimensional, 213–217B
  - of pointers to functions, 15B, 113B
  - of pointers, as substitute for arrays of objects, 20B
  - pointers into, 280B
  - using placement `new` to initialize, 20–21B
- `assert`, and `abort`, 167B
- assignment operators — see *`operator=`*
- assignments
  - in reference-counted value classes, 196B
  - mixed-type, 260B, 261B, 263–265B
  - of pointers and references, 11B
  - partial, 259B, 263–265B
  - through pointers, 259B, 260B
- associative arrays, 236B, 237B
- `auto_ptr`, 49B, 53B, 57B, 58B, 137B, 139B, 162B, 240B, 257B
  - and heap arrays, 48B
  - and object ownership, 183B

  - and pass-by-reference, 165B
  - and pass-by-value, 164B
  - and preventing resource leaks, 48B, 58B
  - assignment of, 162–165B
  - copying of, 162–165B
  - implementation, 291–294B

B

`bad_alloc` class, 70B, 75B

`bad_cast` class, 70B, 261B, 262B

`bad_exception` class, 70B, 77B

`bad_typeid` class, 70B

Barton, John J., 288B

base classes
  - and `catch` clauses, 67B
  - and `delete`, 18B
  - for counting objects, 141–145B

BASIC, 156B, 213B

`basic_string` class, 279B, 280B

`begin` function, 283B

benchmarks, 80B, 110B, 111B

best fit, 67B

Bible, allusion to, 235B

`bitset` template, 4B, 283B

books, recommended, 285–289B

`bool`, 3B, 4B

bugs in this book, reporting, 8B

bypassing
  - constructors, 21B

  - exception-related costs, 79B
  - RTTI information, 122B
  - smart pointer smartness, 171B
  - strong typing, 287B
  - virtual base classes, 122B
  - virtual functions, 122B

C

C
  - dynamic memory allocation, 275B
  - functions and name mangling, 271B
  - linkage, 272B
  - migrating to C++, 286B
  - mixing with C++, 270–276B
  - standard library, 278B

*C Programming Language, The*36B

C-style casts, 12B, 90B

C++
  - dynamic memory allocation, 275B
  - migrating from C, 286B
  - mixing with C, 270–276B
  - standard library — see *standard C++ library*

C++ Programming Language, The, 286B
  - *C++ Programming Style*287B
  - *C++ Report*287B, 289B

C++-style casts, 12–16B
  - approximating, 15–16B

*C/C++ Users Journal*289B

`c_str`, 27B

caching, 94–95B, 98B

callback functions, 74–75B, 79B

Candide, allusion to, 19B

Cargill, Tom, 44B, 287B

Carroll, Martin D., 288B

casts

- C++-style, 12–16B
- C-style, 12B, 90B
- of function pointers, 15B, 242B
- safe, 14B
- to remove `const`ness or `volatile`ness, 13B, 221B

`catch`, 56B

- and inheritance, 67B
- and temporary objects, 65B
- by pointer, 70B
- by reference, 71B
- by value, 70B
- clauses, order of examination, 67–68B
- clauses, vs. virtual functions, 67B see also *pass-by-value, pass-by-reference*,and*pass-by-pointer*

change, designing for, 252–270B

`char*`s, vs. `string` objects, 4B

characters

- Unicode, 279B
- wide, 279B

Clancy — see *Urbano, Nancy L.*

classes

- abstract mixin bases, 154B
- abstract, drawing, 5B

  - adding members at runtime, 287B
  - base — see *base classes*
  - concrete, drawing, 5B
  - derived — see *derived classes*
  - designing — see *design*
  - diagnostic, in the standard library, 66B
  - for registering things, 250B
  - mixin, 154B
  - modifying, and recompilation, 234B, 249B
  - nested, and inheritance, 197B
  - proxy — see *proxy classes*
  - templates for, specializing, 175B
  - transforming concrete into abstract, 266–269B
- cleaning your room, 85B
- client, definition, 7B
- CLOS, 230B
- COBOL, 213B, 272B
- code duplication, 47B, 54B, 142B, 204B, 223B, 224B
- code reuse
  - via smart pointer templates and base classes, 211B
  - via the standard library, 5B
- code, generalizing, 258B
- comma operator — see `operator,`
- committee for C++ standardization— see *ANSI/ISO standardization committee*
- `comp.lang.c++`, 289B
- `comp.lang.c++.moderated`, 289B
- `comp.std.c++`, 290B
- comparing addresses to determine object location, 150–152B
- compilers, lying to, 241B

complex numbers, 279B, 280
concrete classes
    and inheritance, 258–270B
    drawing, 5B
    transforming into abstract, 266–269B
consistency
    among `+`, `=`, and `+=`, etc., 107B
    between built-in and user defined types, 254B
    between prefix and postfix `operator++` and `operator--`, 34B
    between real and virtual copy constructors, 126B
`const` member functions, 89B, 160B, 218B
`const` return types, 33–34B, 101B
`const static` data members, initialization, 140B
`const_cast`, 13B, 14B, 15B, 37B, 90B
`const_iterator` type, 127B, 283B
constant pointers, 55–56B
`const`ness, casting away, 13B, 221B
constructing objects on top of one another, 287B
constructors
    and fully constructed objects, 52B
    and `malloc`, 39B
    and memory leaks, 6B
    and `operator new`, 39B, 149–150B
    and `operator new[]`, 43B
    and references, 165B
    and static initialization, 273B
    as type conversion functions, 27 –, 31B
    bypassing, 21B
    calling directly, 39B

  copy — see *copy constructors*
  default — see *default constructors*
  lazy, 88–90B
  preventing exception-related resource leaks, 50–58B
  private, 130B, 137B, 146B
  protected, 142B
  pseudo — see *pseudo-constructors*
  purpose, 21B
  relationship to `new` operator and `operator new`, 40B
  single-argument, 25B, 27–31B, 177B
  virtual — see *virtual constructors*
contacting this book's author, 8B
containers — see *Standard Template Library*
contexts for object construction, 136B
conventions
  and the STL, 284B
  for I/O operators, 128B
  used in this book, 5–8B
conversion functions — see *type conversion functions*
conversions — see *type conversions*
Coplien, James, 287B
copy constructors, 146B
  and classes with pointers, 200B
  and exceptions, 63B, 68B
  and non-`const` parameters, 165B
  and smart pointers, 205B
  for strings, 86B
  virtual — see *virtual copy constructors*
copying objects

  - and exceptions, 68B
  - static type vs. dynamic type, 63B
  - when throwing an exception, 62 –, 63B
- copy-on-write, 190–194B
- counting object instantiations, 141 –, 145B
- C-style casts, 12B
- ctor, definition, 6B
- customizing memory management, 38–43B

D

- data members
  - adding at runtime, 287B
  - `auto_ptr`, 58B
  - initialization when `const`, 55–56B
  - initialization when static, 140B
  - replication, under multiple inheritance, 118–120B
  - `static`, in templates, 144B
- dataflow languages, 93B
- Davis, Bette, allusion to, 230B
- decrement operator — see *`operator--`*
- default constructors
  - and arrays, 19–21B
  - and templates, 22B
  - and virtual base classes, 22B
  - definition, 19B
  - meaningless, 23B
  - restrictions from, 19–22B
  - when to/not to declare, 19B
- `delete and inheritance`, 18B

- and memory not from `new`, 21B
- and nonvirtual destructors, 256B
- and null pointers, 52B
- and objects, 173B
- and smart pointers, 173B
- and `this`, 145B, 152B, 157B, 197B, 213B
- determining when valid, 152–157B see also `delete` *operator* and *ownership*

`delete` operator, 37B, 41B, 173B
- and `operator delete[]` and destructors, 43B
- and placement `new`, 42B
- and `this`, 145B, 152B, 157B, 197B, 213B

deprecated features, 7B
- access declarators, 144B
- statics at file scope, 246B
- `strstream` class, 279B

`deque` template, 283B

derived classes
- and `catch` clauses, 67B
- and `delete`, 18B
- and `operator=`, 263B
- prohibiting, 137B

design
- and multiple dispatch, 235B
- for change, 252–270B
- of classes, 33B, 133B, 186B, 227B, 258B, 268B
- of function locations, 244B
- of libraries, 110B, 113B, 284B, 286B, 288B
- of templates, 286B
- of virtual function implementation, 248B

  patterns, 123B, 288B

Design and Evolution of C++, The, 278B, 285B

Design Patterns: Elements of Reusable Object-Oriented Software, 288B

Designing and Coding Reusable C++, 288B

destruction, static, 273–275B

destructors

  and `delete`, 256B

  and exceptions, 45B

  and fully constructed objects, 52B

  and `longjmp`, 47B

  and memory leaks, 6B

  and `operator delete[]`, 43B

  and partially constructed objects, 53B

  and smart pointers, 205B

  private, 145B

  protected, 142B, 147B

  pseudo, 145B, 146B

  pure virtual, 195B, 265B

  virtual, 143B, 254–257B

determining whether a pointer can be deleted, 152–157B

determining whether an object is on the heap, 147–157B

diagnostics classes of the standard library, 66B

dispatching — see *multiple dispatch*

distinguishing lvalue and rvalue use of `operator[]`, 87B, 217–223B

`domain_error` class, 66B

double application of increment and decrement, 33B

double-dispatch— see *multiple dispatch*

dtor, definition, 6B

dumb pointers, 159B, 207B

duplication of code, 47B, 54B, 142B, 204B, 223B, 224B
dynamic arrays, 96–98B
dynamic type
    vs. static type, 5–6B
    vs. static type, when copying, 63B
`dynamic_cast`, 6B, 37B, 261–262B
    and null pointer, 70B
    and virtual functions, 14B, 156B
    approximating, 16B
    meaning, 14B
    to get a pointer to the beginning of an object, 155B
    to reference, failed, 70B
    to `void*`, 156B

E

eager evaluation, 86B, 91B, 92B, 94B, 98B
    converting to lazy evaluation, 93B
Edelson, Daniel, 179B
    *Effective C++*5B, 100B, 286B
efficiency
    and assigning smart pointers, 163B
    and benchmarks, 110B
    and cache hit rate, 98B
    and constructors and destructors, 53B
    and copying smart pointers, 163B
    and encapsulation, 82B
    and function return values, 101B
    and inlining, 129B
    and libraries, 110B, 113B

and maintenance, 91B
and multiple inheritance, 118 –, 120B
and object size, 98B
and `operator`s `new` and `delete`, 97B, 113B
and paging behavior, 98B
and pass-by-pointer, 65B
and pass-by-reference, 65B
and pass-by-value, 65B
and profiling, 84–85B, 93B
and reference counting, 183B, 211B
and system calls, 97B
and temporary objects, 99–101B
and tracking heap allocations, 153B
and virtual functions, 113–118B
and vptrs, 116B, 256B
and vtbls, 114B, 256B
caching, 94–95B, 98B
class statics vs. function statics, 133B
cost amortization, 93–98B
implications of meaningless default constructors, 23B
iostreams vs. stdio, 110–112B
locating bottlenecks, 83B
manual methods vs. language features, 122B
of exception-related features, 64B, 70B, 78–80B
of prefix vs. postfix increment and decrement, 34B
of stand-alone operators vs. assignment versions, 108B
prefetching, 96–98B
reading vs. writing referencecounted objects, 87B, 217B
space vs. time, 98B
summary of costs of various language features, 121B

  - virtual functions vs. manual methods, 121B, 235B
  - vs. syntactic convenience, 108B see also *optimization*

- Ellis, Margaret A., 285B, 288B
- emulating features— see *approximating*
- encapsulation
  - allowing class implementations to change, 207B
  - and efficiency, 82B
- `end` function, 283B
- enums
  - and overloading operators, 277B
  - as class constants, 141B
- evaluation
  - converting eager to lazy, 93B
  - eager, 86B, 91B, 92B, 94B, 98B
  - lazy, 85–93B, 94B, 98B, 191B, 219B
  - over-eager, 94–98B
  - short-circuit, 35B, 36B
- example uses
  - `__cplusplus`, 273B
  - `auto_ptr`, 48B, 57B, 138B, 164B, 165B, 240B, 247B, 257B
  - `const` pointers, 55B
  - `const_cast`, 13B, 90B, 221B
  - `dynamic_cast`, 14B, 155B, 243B, 244B, 261B, 262B
  - exception specifications, 70B, 73B, 74B, 75B, 77B
  - `explicit`, 29B, 291B, 293B
  - `find` function, 283B
  - implicit type conversion operators, 25B, 26B, 49B, 171B, 175B, 219B, 225B
  - in-class initialization of `const static` members, 140B
  - `list` template, 51B, 124B, 154B , 283B

`make_pair` template, 247B
`map` template, 95B, 238B, 245B
member templates, 176B, 291B, 292B, 293B
`mutable`, 88B
`namespace`, 132B, 245B, 246B, 247B
nested class using inheritance, 197B
`operator delete`, 41B, 155B
`operator delete[]`, 21B
`operator new`, 41B, 155B
`operator new[]`, 21B
`operator&`, 224B
`operator->*` (built-in), 237B
`pair` template, 246B
placement `new`, 21B, 40B
pointers to member functions, 236B, 238B
pure virtual destructors, 154B, 194B
reference data member, 219B
refined return type of virtual functions, 126B, 260B
`reinterpret_cast`, 15B
`setiosflags`, 111B
`setprecision`, 111B
`setw`, 99B, 111B
Standard Template Library, 95B , 125B, 127B, 155B, 238B, 247B, 283B, 284B
`static_cast`, 12B, 18B, 21B, 28B, 29B, 231B
`typeid`, 231B, 238B, 245B
`using` declarations, 133B, 143B
`vector` template, 11B
`exception` class, 66B, 77B
exception specifications, 72–78B

  - advantages, 72B
  - and callback functions, 74–75B
  - and layered designs, 77B
  - and libraries, 76B, 79B
  - and templates, 73–74B
  - checking for consistency, 72B
  - cost of, 79B
  - mixing code with and without, 73B, 75B
- `exception::what`, 70B, 71B
- exceptions, 287B
  - and destructors, 45B
  - and `operator new`, 52B
  - and type conversions, 66–67B
  - and virtual functions, 79B
  - causing resource leaks in constructors, 52B
  - choices for passing, 68B
  - disadvantages, 44B
  - efficiency, 63B, 65B, 78–80B
  - mandatory copying, 62–63B
  - modifying `throw` expressions, 63B
  - motivation, 44B
  - optimization, 64B
  - recent revisions to, 277B
  - rethrowing, 64B
  - specifications — see *exception specifications*
  - standard, 66B, 70B
  - unexpected — see *unexpected exceptions*
  - use of copy constructor, 63B
  - use to indicate common conditions, 80B

  vs. `setjmp` and `longjmp`, 45B see also *`catch`*, *`throw`*

explicit 28–31B, 227B, 294B

`extern "C"`, 272–273B

F

fake `this`, 89B

`false`, 3B

  *Felix the Cat*123B

fetch and increment, 32B

fetching, lazy, 87–90B

`find` function, 283B

first fit, 67B

fixed-format I/O, 112B

Forth, 271B

FORTRAN, 213B, 215B, 271B, 288B

`free`, 42B, 275B

French, gratuitous use of, 177B, 185B

friends, avoiding, 108B, 131B

`fstream` class, 278B

FTP site

  for this book, 8B, 287B

fully constructed objects, 52B

function call semantics, 35–36B

functions

  adding at runtime, 287B

  C, and name mangling, 271B

  callback, 74–75B, 79B

  for type conversions, 25–31B

  inline, in this book, 7B

member — see *member functions*
member template — see *member templates*
return types — see *return types*
virtual — see *virtual functions*
future tense programming, 252–258B

G

Gamma, Erich, 288B
garbage collection, 183B, 212B
generalizing code, 258B
German, gratuitous use of, 31B
global overloading of `operator new/delete`, 43B, 153B
GUI systems, 49B, 74–75B

H

Hamlet, allusion to, 22B, 70B, 252B
heap objects — see *objects*
Helm, Richard, 288B
heuristic for vtbl generation, 115B

I

identifying abstractions, 267B, 268B
idioms, 123B
Iliad, Homer's, 87B
implementation
of `+` in terms of `+=`, etc., 107B
of libraries, 288B

- of multiple dispatch, 230–251B
- of operators `++` and `--`, 34B
- of pass-by-reference, 242B
- of pure virtual functions, 265B
- of references, 242B
- of RTTI, 120–121B
- of virtual base classes, 118–120B
- of virtual functions, 113–118B

implicit type conversion operators— see *type conversion operators*

implicit type conversions— see *type conversions*

increment and fetch, 32B

increment operator— see *`operator++`*

indexing, array
- and inheritance, 17–18B
- and pointer arithmetic, 17B

inheritance
- and abstract classes, 258–270B
- and `catch` clauses, 67B
- and concrete classes, 258–270B
- and `delete`, 18B
- and emulated vtbls, 248–249B
- and libraries, 269–270B
- and nested classes, 197B
- and `operator delete`, 158B
- and `operator new`, 158B
- and private constructors and destructors, 137B, 146B
- and smart pointers, 163B, 173 –, 179B
- and type conversions of exceptions, 66B
- multiple — see *multiple inheritance*

- private, 143B

initialization
- demand-paged, 88B
- of arrays via placement `new`, 20–21B
- of `const` pointer members, 55–56B
- of `const static` members, 140B
- of emulated vtbls, 239–244B, 249 –, 251B
- of function statics, 133B
- of objects, 39B, 237B
- of pointers, 10B
- of references, 10B
- order, in different translation units, 133B
- static, 273–275B

inlining
- and “virtual” non-member functions, 129
- and function statics, 134B
- and the return value optimization, 104B
- and vtbl generation, 115B
- in this book, 7B

instantiations, of templates, 7B

internal linkage, 134B

Internet site
- for free STL implementation, 4B
- for this book, 8B, 287B

`invalid_argument` class, 66B

`iostream` class, 278B

iostreams, 280B
- and fixed-format I/O, 112B
- and `operator!`, 170B

  - conversion to `void*`, 168B
  - vs. stdio, 110–112B

ISO standardization committee— see *ANSI/ISO standardization committee*

iterators, 283B

  - and `operator->`, 96B
  - vs. pointers, 282B, 284B see also *Standard Template Library*

J

Japanese, gratuitous use of, 45B

Johnson, Ralph, 288B

K

Kernighan, Brian W., 36B

Kirk, Captain, allusion to, 79B

L

language lawyers, 276B, 290B

Latin, gratuitous use of, 203B, 252B

lazy construction, 88–90B

lazy evaluation, 85–93B, 94B, 191B , 219B

  - and object dependencies, 92B
  - conversion from eager, 93B
  - when appropriate, 93B, 98B

lazy fetching, 87–90B

leaks, memory — see *memory leaks*

leaks, resource— see *resource leaks*

`length_error` class, 66B

`lhs`, definition, 6B

libraries
  andexceptionspecifications, 75B, 76B, 79B
  design and implementation, 110B, 113B, 284B, 286B, 288B
  impact of modification, 235B
  inheriting from, 269–270B
library, C++ standard — see *standard C++ library*
lifetime of temporary objects, 278B
limitations on type conversion sequences, 29B, 31B, 172B , 175B, 226B
limiting object instantiations, 130–145B
linkage
  C, 272B
  internal, 134B
linkers, and overloading, 271B
Lisp, 93B, 230B, 271B
`list` template, 4B, 51B, 124B, 125B, 154B, 283B
locality of reference, 96B, 97B
localization, support in standard C++ library, 278B
`logic_error` class, 66B
`longjmp` and destructors, 47B
  and `setjmp`, vs. exceptions, 45B
LSD, 287B
lvalue, definition, 217B
lying to compilers, 241B

M

magazines, recommended, 289B
`main`, 251B, 273B, 274B
maintenance, 57B, 91B, 107B, 179B, 211B, 227B, 253B, 267B , 270B, 273B
  and RTTI, 232B

`make_pair` template, 247B
`malloc`, 39B, 42B, 275B
    and constructors, 39B
    and `operator new`, 39B
`map` template, 4B, 95B, 237B, 246B , 283B
member data— see *data members*
member functions
    and compatibility of C++ and C structs, 276B
    `const`, 89B, 160B, 218B
    invocation through proxies, 226B
    pointers to, 240B
member initialization lists, 58B
    and `?:` vs. `if/then`, 56B
    and `try` and `catch`, 56B
member templates, 165B
    and assigning smart pointers, 180B
    and copying smart pointers, 180B
    approximating, 294B
    for type conversions, 175–179B
    portability of, 179B
memory allocation, 112B
    for `basic_string` class, 280B
    for heap arrays, 42–43B
    for heap objects, 38B
    in C++ vs. C, 275B
memory leaks, 6B, 7B, 42B, 145B see also *resource leaks*
memory management, customizing, 38–43B
memory values, after calling `operator new`, 38B
memory, shared, 40B
memory-mapped I/O, 40B

message dispatch— see *multiple dispatch*
migrating from C to C++, 286B
mixed-type assignments, 260B, 261B
  prohibiting, 263–265B
mixed-type comparisons, 169B
mixin classes, 154B
mixing code
  C++ and C, 270–276B
  with and without exception specifications, 75B
multi-dimensional arrays, 213–217B
multi-methods, 230B
multiple dispatch, 230–251B
multiple inheritance, 153B
  and object addresses, 241B
  and vptrs and vtbls, 118–120B
`mutable`, 88–90B

N

Nackman, Lee R., 288B
`name` function, 238B
name mangling, 271–273B
named objects, 109B
  and optimization, 104B
  vs. temporary objects, 109B
namespaces, 132B, 144B
  and the standard C++ library, 280B
  `std`, 261B
  unnamed, 246B, 247B
nested classes, and inheritance, 197B

new language features, summary, 277B
`new` operator, 37B, 38B, 42B
    and `bad_alloc`, 75B
    and `operator new` and constructors, 39B, 40B
    and `operator new[]` and constructors, 43B
`new` , placement — *see* placement `new` newsgroups, recommended, 289B
Newton, allusion to, 41B
non-member functions, acting virtual, 128–129B
null pointers
    and `dynamic_cast`, 70B
    and `strlen`, 35B
    and the STL, 281B
    deleting, 52B
    dereferencing, 10B
    in smart pointers, 167B
    testing for, 10B
null references, 9–10B
numeric applications, 90B, 279B

O

objects
    addresses, 241B
    allowing exactly one, 130–134B
    and virtual functions, 118B
    as function return type, 99B
    assignments through pointers, 259B, 260B
    constructing on top of one another, 287B
    construction, lazy, 88–90B
    contexts for construction, 136B

  - copying, and exceptions, 62–63B, 68B
  - counting instantiations, 141–145B
  - deleting, 173B
  - determining location via address comparisons, 150–152B
  - determining whether on the heap, 147–157B
  - initialization, 39B, 88B, 237B
  - limiting the number of, 130–145B
  - locations, 151B
  - memory layout diagrams, 116B, 119B, 120B, 242B
  - modifying when thrown, 63B
  - named — see *named objects*
  - ownership, 162B, 163–165B, 183B
  - partially constructed, 53B
  - preventing instantiations, 130B
  - prohibiting from heap, 157–158B
  - proxy — see *proxy objects*
  - restricting to heap, 145–157B
  - size, and cache hit rate, 98B
  - size, and paging behavior, 98B
  - static — see *static objects*
  - surrogate, 217B
  - temporary — see *temporary objects*
  - unnamed — see *temporary objects*
  - using `dynamic_cast` to find the beginning, 155B
  - using to prevent resource leaks, 47–50B, 161B
  - vs. pointers, in classes, 147B
- *On Beyond Zebra*, allusion to, 168B
- `operator delete`, 37B, 41B, 84B, 113B, 173B
  - and efficiency, 97B
  - and inheritance, 158B

`operator delete[]`, 37B, 84B
  and `delete` operator and destructors, 43B
  private, 157B
`operator new`, 37B, 38B, 69B, 70B, 84B, 113B, 149B
  and `bad_alloc`, 75B
  and constructors, 39B, 149–150B
  and efficiency, 97B
  and exceptions, 52B
  and inheritance, 158B
  and `malloc`, 39B
  and `new` operator and constructors, 40B
  calling directly, 39B
  overloading at global scope, 43B, 153B
  private, 157B
  values in memory returned from, 38B
`operator new[]`, 37B, 42B, 84B, 149B
  and `bad_alloc`, 75B
  and `new` operator and constructors, 43B
  private, 157B
operator overloading, purpose, 38B
`operator void*`, 168–169B
`operator!`, 37B
  in iostream classes, 170B
  in smart pointers classes, 169B
`operator!=`, 37B
`operator%`, 37B
`operator%=`, 37B
`operator&`, 37B, 74B, 223B
`operator&&`, 35–36B, 37B

`operator&=`, 37B

`operator()`, 37B, 215B

`operator*`, 37B, 101B, 103B, 104B, 107B

  and null smart pointers, 167B

  and STL iterators, 96B

  as `const` member function, 160B

`operator*=`, 37B, 107B, 225B

`operator+`, 37B, 91B, 100B, 107B, 109B

  template for, 108B

`operator++`, 31–34B, 37B, 225B

  double application of, 33B

  prefix vs. postfix, 34B

`operator+=`, 37B, 107B, 109B, 225B

`operator,`, 36–37B

`operator-`, 37B, 107B

  template for, 108B

`operator-=`, 37B, 107

`operator->`, 37B

  and STL iterators, 96B

  as `const` member function, 160B

`operator->*`, 37B

`operator--`, 31–34B, 37B, 225B

  double application of, 33B

  prefix vs. postfix, 34B

`operator.`, 37B

  and proxy objects, 226B

`operator.*`, 37B

`operator/`, 37B, 107B

`operator/=`, 37B, 107B

`operator::`, 37B

`operator<`, 37B
`operator<<`, 37B, 112B, 129B
  why a member function, 128B
`operator<<=`, 37B, 225B
`operator<=`, 37B
`operator=`, 37B, 107B, 268B
  and classes with pointers, 200B
  and derived classes, 263B
  and inheritance, 259–265B
  and mixed-type assignments, 260B, 261B , 263–265B
  and non-`const` parameters, 165B
  and partial assignments, 259B , 263–265B
  and smart pointers, 205B
  virtual, 259–262B
`operator==`, 37B
`operator>`, 37B
`operator>=`, 37B
`operator>>`, 37B
`operator>>=`, 37B
`operator?:`, 37B, 56B
`operator[]`, 11B, 37B, 216B
  `const` vs. non-`const`, 218B
  distinguishing lvalue and rvalue use, 87B, 217–223B
`operator[][]`, 214B
`operator^`, 37B
`operator^=`, 37B
`operator|`, 37B
`operator|=`, 37B
`operator||`, 35–36B, 37B

`operator~`, 37B
operators
  implicit type conversion — see *type conversion operators*
  not overloadable, 37B
  overloadable, 37B
  returning pointers, 102B
  returning references, 102B
  stand-alone vs. assignment versions, 107–110B
optimization
  and profiling data, 84B
  and `return` expressions, 104B
  and temporary objects, 104B
  of exceptions, 64B
  of reference counting, 187B
  of vptrs under multiple inheritance, 120B
  return value — see *return value optimization*
  via `valarray` objects, 279B see also *efficiency*
order of examination of `catch` clauses, 67–68B
Ouija boards, 83B
`out_of_range` class, 66B
over-eager evaluation, 94–98B
`overflow_error` class, 66B
overloadable operators, 37B
overloading
  and enums, 277B
  and function pointers, 243B
  and linkers, 271B
  and user-defined types, 106B
  `operator new/delete` at global scope, 43B, 153B
  resolution of function calls, 233B

    restrictions, 106B
    to avoid type conversions, 105 –, 107B
ownership of objects, 162B, 183B
    transferring, 163–165B

P

`pair` template, 246B
parameters
    passing, vs. throwing exceptions, 62–67B
    unused, 33B, 40B
partial assignments, 259B, 263–265B
partial computation, 91B
partially constructed objects, and destructors, 53B
pass-by-pointer, 65B
pass-by-reference
    and `auto_ptr`s, 165B
    and `const`, 100B
    and temporary objects, 100B
    and the STL, 282B
    and type conversions, 100B
    efficiency, and exceptions, 65B
    implementation, 242B
pass-by-value
    and `auto_ptr`s, 164B
    and the STL, 282B
    and virtual functions, 70B
    efficiency, and exceptions, 65B
passing exceptions, choices, 68B
patterns, 123B, 288B

Pavlov, allusion to, 81B
performance — see *efficiency*
placement `new`, 39–40B
  and array initialization, 20–21B
  `and delete` operator, 42B
pointer arithmetic
  and array indexing, 17B
  and inheritance, 17–18B
pointers
  and object assignments, 259B, 260B
  and proxy objects, 223B
  and virtual functions, 118B
  as parameters — see *pass-by-pointer*
  assignment, 11B
  constant, 55B
  dereferencing when null, 10B
  determining whether they can be deleted, 152–157B
  dumb, 159B
  implications for copy constructors and assignment operators, 200B
  initialization, 10B, 55–56B
  into arrays, 280B
  null — see *null pointers*
  replacing dumb with smart, 207B
  returning from operators, 102B
  smart — see *smart pointers*
  testing for nullness, 10B
  to functions, 15B, 241B, 243B
  to member functions, 240B
  vs. iterators, 284B

  - vs. objects, in classes, 147B
  - vs. references, 9–11B
  - when to use, 11B
- polymorphism, definition, 16B
- *Poor Richard's Almanac*, allusion to, 75B
- portability
  - and non-standard functions, 275B
  - of casting function pointers, 15B
  - of determining object locations, 152B, 158B
  - of `dynamic_cast` to `void*`, 156B
  - of member templates, 179B
  - of passing data between C++ and C, 275B
  - of `reinterpret_cast`, 14B
  - of static initialization and destruction, 274B
- prefetching, 96–98B
- preventing object instantiation, 130B
- principle of least astonishment, 254B
- `printf`, 112B
- `priority_queue` template, 283B
- private inheritance, 143B
- profiling — see *program profiling*
- program profiling, 84–85B, 93B , 98B, 112B, 212B
- programming in the future tense, 252–258B
- protected constructors and destructors, 142B
- proxy classes, 31B, 87B, 190B, 194B, 213–228B
  - definition, 217B
  - limitations, 223–227B see also *proxy objects*
- proxy objects
  - and `++`, `--`, `+=`, etc., 225B

  - and member function invocations, 226B
  - and `operator.`, 226B
  - and pointers, 223B
  - as temporary objects, 227B
  - passing to non-`const` reference parameters, 226B see also *proxy classes*
- pseudo-constructors, 138B, 139B, 140B
- pseudo-destructors, 145B, 146B
- pure virtual destructors, 195B, 265B
- pure virtual functions— see *virtual functions*
- Python, Monty, allusion to, 62B

Q

- `queue` template, 4B, 283B

R

- `range_error` class, 66B
- recommended reading
  - books, 285–289B
  - magazines, 289B
  - on exceptions, 287B
  - Usenet newsgroups, 289B
- recompilation, impact of, 234B, 249B
- reference counting, 85–87B, 171B , 183–213B, 286B
  - and efficiency, 211B
  - and read-only types, 208–211B
  - and shareability, 192–194B
  - assignments, 189B
  - automating, 194–203B

- base class for, 194–197B
- constructors, 187–188B
- cost of reads vs. writes, 87B, 217B
- design diagrams, 203B, 208B
- destruction, 188B
- implementation of `String` class, 203–207B
- `operator[]`, 190–194B
- optimization, 187B
- pros and cons, 211–212B
- smart pointer for, 198–203B
- when appropriate, 212B

references
- and constructors, 165B
- and virtual functions, 118B
- as `operator[]` return type, 11B
- as parameters — see *pass-by-reference*
- assignment, 11B
- implementation, 242B
- mandatory initialization, 10B
- null, 9–10B
- returning from operators, 102B
- to locals, returning, 103B
- vs. pointers, 9–11B
- when to use, 11B

refined return type of virtual functions, 126B

`reinterpret_cast`, 14–15B, 37B, 241B

relationships
- among `delete` operator, `operator delete`, and destructors, 41B
- among `delete` operator, `operator delete[]` , and destructors, 43B

  - among `new` operator, `operator new`, and constructors, 40B
  - among `new` operator, `operator new[]`, and constructors, 43B
  - among `operator+`, `operator=`, and `operator+=`, 107B
  - between `operator new` and `bad_alloc`, 75B
  - between `operator new[]` and `bad_alloc`, 75B
  - between `terminate` and `abort`, 72B
  - between the `new` operator and `bad_alloc`, 75B
  - between `unexpected` and `terminate`, 72B
- replication of code, 47B, 54B, 142B, 204B, 223B, 224B
- replication of data under multiple inheritance, 118–120B
- reporting bugs in this book, 8B
- resolution of calls to overloaded functions, 233B
- resource leaks, 46B, 52B, 69B, 102B, 137B, 149B, 173B, 240B
  - and exceptions, 45–58B
  - and smart pointers, 159B
  - definition, 7B
  - in constructors, 52B, 53B
  - preventing via use of objects, 47–50B, 58B, 161B
  - vs. memory leaks, 7B
- restrictions on classes with default constructors, 19–22
- rethrowing exceptions, 64B
- `return` expression, and optimization, 104B
- return types
  - and temporary objects, 100–104B
  - `const`, 33–34B, 101B
  - objects, 99B
  - of `operator--`, 32B
  - of `operator++`, 32B
  - of `operator[]`, 11B

  - of smart pointer dereferencing operators, 166B
  - of virtual functions, 126B
  - references, 103B

return value optimization, 101 –, 104B, 109B

reuse — see *code reuse*

`rhs`, definition, 6B

rights and responsibilities, 213B

Ritchie, Dennis M., 36B

Romeo and Juliet, allusion to, 166B

RTTI, 6B, 261–262B
  - and maintenance, 232B
  - and virtual functions, 120B, 256B
  - and vtbls, 120B
  - implementation, 120–121B
  - vs. virtual functions, 231–232B

runtime type identification— see *RTTI*

`runtime_error` class, 66B

rvalue, definition, 217B

S

safe casts, 14B

Scarlet Letter, The, allusion to, 232B

*Scientific and Engineering C++*288B

semantics of function calls, 35–36B

sequences of type conversions, 29B, 31B, 172B, 175B, 226B

`set` template, 4B, 283B

`set_unexpected` function, 76B

`setiosflags`, example use, 111B

`setjmp` and `longjmp` , vs. exceptions, 45B `setprecision`, example use, 111B

`setw`, example uses, 99B, 111B

shared memory, 40B

sharing values, 86B see also *reference counting*

short-circuit evaluation, 35B, 36B

single-argument constructors— see *constructors*

`sizeof`, 37B

slicing problem, 70B, 71B

smart pointers, 47B, 90B, 159 –, 182B, 240B, 282B

- and `const`, 179–182B
- and distributed systems, 160 –, 162B
- and inheritance, 163B, 173–179B
- and member templates, 175 –, 182B
- and resource leaks, 159B, 173B
- and virtual constructors, 163B
- and virtual copy constructors, 202B
- and virtual functions, 166B
- assignments, 162–165B, 180B
- construction, 162B
- conversion to dumb pointers, 170–173B
- copying, 162–165B, 180B
- debugging, 182B
- deleting, 173B
- destruction, 165–166B
- for reference counting, 198–203B
- `operator*`, 166–167B
- `operator->`, 166–167B
- replacing dumb pointers, 207B
- testing for nullness, 168–170B, 171B

Spanish, gratuitous use of, 232B

`sqrt` function, 65B
stack objects — see *objects*
`stack` template, 4B, 283B
standard C library, 278B
standard C++ library, 1B, 4–5B, 11B, 48B, 51B, 280B
  and code reuse, 5B
  diagnostics classes, 66B
  summary of features, 278–279B
  use of templates, 279–280B see also *Standard Template Library*
Standard Template Library, 4–5B , 95–96B, 280–284B
  and pass-by-reference, 282B
  and pass-by-value, 282B
  conventions, 284B
  example uses — see *example uses*
  extensibility, 284B
  free implementation, 4B
  iterators and `operator->`, 96B
standardization committee— see *ANSI/ISO standardization committee*
Star Wars, allusion to, 31B
static destruction, 273–275B
static initialization, 273–275B
static objects, 151B
  and inlining, 134B
  at file scope, 246B
  in classes vs. in functions, 133 –, 134B
  in functions, 133B, 237B
  when initialized, 133B
static type vs. dynamic type, 5–6B
  when copying, 63B

`static_cast`, 13B, 14B, 15B, 37B
`std` namespace, 261B
    and standard C++ library, 280B
stdio, vs. iostreams, 110–112B
STL — see *Standard Template Library*
`strdup`, 275B
`string` class, 27B, 279–280B
    `c_str` member function, 27B
    destructor, 256B
`String` class — see *reference counting*
`string` objects, vs. `char*`s, 4B
`stringstream` class, 278B
`strlen`, and null pointer, 35B
strong typing, bypassing, 287B
Stroustrup, Bjarne, 285B, 286B
`strstream` class, 278B
structs
    compatibility between C++ and C, 276B
    private, 185B
summaries
    of efficiency costs of various language features, 121B
    of new language features, 277B
    of standard C++ library, 278–279B
suppressing
    type conversions, 26B, 28–29B
    warnings for unused parameters, 33B, 40B
Surgeon General's tobacco warning, allusion to, 288B
surrogates, 217B
Susann, Jacqueline, 228B
system calls, 97B

T

templates, 286B, 288B

- and “>>”, vs. “> >”, 29B
- and default constructors, 22B
- and exception specifications, 73 –, 74B
- and pass-by-reference, 282B
- and pass-by-value, 282B
- and static data members, 144B
- for `operator+` and `operator`, 108B
- in standard C++ library, 279 –, 280B
- member — see *member templates*
- recent extensions, 277B
- specializing, 175B
- vs. template instantiations, 7B

temporary objects, 34B, 64B, 98 –, 101B, 105B, 108B, 109B

- and efficiency, 99–101B
- and exceptions, 68B
- and function return types, 100–104B
- and optimization, 104B
- and pass-by-reference, 100B
- and type conversions, 99–100B
- catching vs. parameter passing, 65B
- eliminating, 100B, 103–104B
- lifetime of, 278B
- vs. named objects, 109B

`terminate`, 72B, 76B

terminology used in this book, 5–8B

`this`, deleting, 145B, 152B, 157B, 197B, 213B

`throw by pointer`, 70B

- cost of executing, 63B, 79B
- modifying objects thrown, 63B
- to rethrow current exception, 64B
- vs. parameter passing, 62–67B see also *`catch`*

transforming concrete classes into abstract, 266–269B

`true`, 3B

`try` blocks, 56B, 79B

type conversion functions, 25–31B

- valid sequences of, 29B, 31B, 172B, 175B, 226B

type conversion operators, 25B, 26–27B, 49B, 168B

- and smart pointers, 175B
- via member templates, 175–179B

type conversions, 66B, 220B, 226B

- and exceptions, 66–67B
- and function pointers, 241B
- and pass-by-reference, 100B
- and temporary objects, 99–100B
- avoiding via overloading, 105–107B
- implicit, 66B, 99B
- suppressing, 26B, 28–29B
- via implicit conversion operators, 25B, 26–27B, 49B
- via single-argument constructors, 27–31B

type errors, detecting at runtime, 261–262B

type system, bypassing, 287B

`type_info` class, 120B, 121B, 261B

- `name` member function, 238B

`typeid`, 37B, 120B, 238B

types, static vs. dynamic, 5–6B

    when copying, 63B

U

undefined behavior
    calling `strlen` with null pointer, 35B
    deleting memory not returned by `new`, 21B
    deleting objects twice, 163B, 173B
    dereferencing null pointers, 10B, 167B
    dereferencing pointers beyond arrays, 281B
    mixing `new`/`free` or `malloc` /
    `delete`, 275B
    `unexpected`, 72B, 74B, 76B, 77B, 78B
unexpected exceptions, 70B
    handling, 75–77B
    replacing with other exceptions, 76B see also *`unexpected`*
Unicode, 279B
`union` , using to avoid unnecessary data, 182B
unnamed namespaces, 246B, 247B
unnamed objects — see *temporary objects*
unused parameters, suppressing warnings about, 33B, 40B
Urbano, Nancy L. — see *Clancy*
URL for this book, 8B, 287B
use counting, 185B see also *reference counting*
useful abstractions, 267B
Usenet newsgroups, recommended, 289B
user-defined conversion functions— see *type conversion functions*
user-defined types
    and overloaded operators, 106B
    consistency with built-ins, 254B

`using` declarations, 133B, 143B

V

`valarray` class, 279B, 280B
values, as parameters — see *pass-by-value*
`vector` template, 4B, 11B, 22B, 283B
virtual base classes, 118–120B, 154B
    and default constructors, 22B
    and object addresses, 241B
virtual constructors, 46B, 123–127B
    and smart pointers, 163B
    definition, 126B
    example uses, 46B, 125B
virtual copy constructors, 126–127B
    and smart pointers, 202B
virtual destructors, 143B, 254–257B
    and `delete`, 256B see also *pure virtual destructors*
virtual functions
    “demand-paged”, 253B
    and `dynamic_cast`, 14B, 156B
    and efficiency, 113–118
    and exceptions, 79B
    and mixed-type assignments, 260B, 261B
    and pass/`catch`-by-reference, 72B
    and pass/`catch`-by-value, 70B
    and RTTI, 120B, 256B
    and smart pointers, 166B
    design challenges, 248B
    efficiency, 118B

  implementation, 113–118B
  pure, 154B, 265B
  refined return type, 126B, 260B
  vs. `catch` clauses, 67B
  vs. RTTI, 231–232B
  vtbl index, 117B
“virtual” non-member functions, 128–129B
virtual table pointers — see *vptrs*
virtual tables — see *vtbls*
Vlissides, John, 288B
`void*`, `dynamic_cast` to, 156B
`volatile`ness, casting away, 13B
vptrs, 113B, 116B, 117B, 256B
  and efficiency, 116B
  effective overhead of, 256B
  optimization under multiple inheritance, 120B
vtbls, 113–116B, 117B, 121B, 256B
  and abstract classes, 115B
  and `inline` virtual functions, 115B
  and RTTI, 120B
  emulating, 235–251B
  heuristic for generating, 115B

W

warnings, suppressing
  for unused parameters, 33B, 40B
`what` function, 70B, 71B
wide characters, 279B
World Wide Web URL for this book, 8B, 287B

Y

Yiddish, gratuitous use of, 32B

Index of Example Classes, Functions, and Templates

Classes and Class Templates

`AbstractAnimal`, 264B

`ALA`, 46B

`Animal`, 258B, 259B, 263B, 265B

`Array`, 22B, 27B, 29B, 30B, 225B

`Array::ArraySize`, 30B

`Array::Proxy`, 225B

`Array2D`, 214B, 215B, 216B

`Array2D::Array1D`, 216B

`Asset`, 147B, 152B, 156B, 158B

`Asteroid`, 229B

`AudioClip`, 50B

`B`, 255B

`BalancedBST`, 16B

`BookEntry`, 51B, 54B, 55B, 56B, 57B

`BST`, 16B

`C1`, 114B

`C2`, 114B

`CallBack`, 74B

`CantBeInstantiated`, 130B

`Cassette`, 174B

`CasSingle`, 178B

`CD`, 174B

`Chicken`, 259B, 260B, 263B, 265B

`CollisionMap`, 249B

`CollisionWithUnknownObject`, 231B

`ColorPrinter`, 136B

`Counted`, 142B

`CPFMachine`, 136B

`D`, 255B

`DataCollection`, 94B

`DBPtr`, 160B, 171B

`DynArray`, 96B

`EquipmentPiece`, 19B, 23B

`FSA`, 137B

`GameObject`, 229B, 230B, 233B, 235B, 242B

`Graphic`, 124B, 126B, 128B, 129B

`HeapTracked`, 154B

`HeapTracked::MissingAddress`, 154B

`Image`, 50B

`Kitten`, 46B

`LargeObject`, 87B, 88B, 89B

`Lizard`, 259B, 260B, 262B, 263B, 265B

`LogEntry`, 161B

`Matrix`, 90B

`MusicProduct`, 173B

`Name`, 25B

`NewsLetter`, 124B, 125B, 127B

`NLComponent`, 124B, 126B, 128B, 129B

`NonNegativeUPNumber`, 146B, 147B, 158B

`PhoneNumber`, 50B

`Printer`, 130B, 132B, 135B, 138B, 140B, 141B, 143B, 144B

`Printer::TooManyObjects`, 135B, 138B, 140B

`PrintingStuff::Printer`, 132B

`PrintJob`, 130B, 131B

`Puppy`, 46B

`Rational`, 6B, 25B, 26B, 102B, 107B, 225B

`RCIPtr`, 209B

`RCObject`, 194B, 204B

`RCPtr`, 199B, 203B

`RCWidget`, 210B

`RegisterCollisionFunction`, 250B

`Satellite`, 251B

`Session`, 59B, 77B

`SmartPtr`, 160B, 168B, 169B, 176B, 178B, 181B

`SmartPtr<Cassette>`, 175B

`SmartPtr<CD>`, 175B

`SmartPtrToConst`, 181B

`SpaceShip`, 229B, 230B, 233B, 235B, 236B, 238B, 239B, 240B, 243B

`SpaceStation`, 229B

`SpecialWidget`, 13B, 63B

`SpecialWindow`, 269B

`String`, 85B, 183B, 186B, 187B, 188B, 189B, 190B, 193B, 197B, 198B, 200B, 201B, 204B, 218B, 219B, 224B

`String::CharProxy`, 219B, 224B

`String::SpecialStringValue`, 201B

`String::StringValue`, 186B, 193B, 197B, 200B, 201B, 204B

`TextBlock`, 124B, 126B, 128B , 129B

`Tuple`, 161B, 170B

`TupleAccessors`, 172B

`TVStation`, 226B

`UnexpectedException`, 76B

`UPInt`, 32B, 105B

`UPNumber`, 146B, 147B, 148B, 157B, 158B

`UPNumber::HeapConstraintViolation`, 148B

`Validation_error`, 70B

`Widget`, 6B, 13B, 40B, 61B, 63B, 210B

`Window`, 269B

`WindowHandle`, 49B

Functions and Function Templates

`AbstractAnimal::~AbstractAnimal`, 264B

`operator=`, 264B

`ALA::processAdoption`, 46B

`allocateSomeObjects`, 151B

`Animal::operator=`, 258B, 259B, 263B, 265B

`Array::`
- `Array`, 22B, 27B, 29B, 30B
- `operator[]`, 27B

`Array::ArraySize::`
- `ArraySize`, 30B
- `size`, 30B

`Array<T>::Proxy::`
- `operator T`, 225B
- `operator=`, 225B
- `Proxy`, 225B

`Array2D::`
- `Array2D`, 214B
- `operator()`, 215B
- `operator[]`, 216B

`Asset::`
- `~Asset`, 147B
- `Asset`, 147B, 158B

`asteroidShip`, 245B

`asteroidStation`, 245B, 250B

`AudioClip::AudioClip`, 50B

BookEntry::

- ~BookEntry, 51B, 55B, 58B
- BookEntry, 51B, 54B, 55B, 56B, 58B
- cleanup, 54B, 55B
- initAudioClip, 57B
- initImage, 57B

C1::

- ~C1, 114B
- C1, 114B
- f1, 114B
- f2, 114B
- f3, 114B
- f4, 114B

C2::

- ~C2, 114B
- C2, 114B
- f1, 114B
- f5, 114B

CallBack::

- CallBack, 74B
- makeCallBack, 74B

callBackFcn1, 75B

callBackFcn2, 75B

CantBeInstantiated::

- CantBeInstantiated, 130B

Cassette::

- Cassette, 174B
- displayTitle, 174B
- play, 174B

CD::

- CD, 174B

  - `displayTitle`, 174B
  - `play`, 174B
- `checkForCollision`, 229B
- `Chicken::operator=`, 259B, 260B, 263B, 265B
- `CollisionMap::`
  - `addEntry`, 249B
  - `CollisionMap`, 249B
  - `lookup`, 249B
  - `removeEntry`, 249B
  - `theCollisionMap`, 249B
- `CollisionWithUnknownObject::`
  - `CollisionWithUnknownObject`, 231B
- `constructWidgetInBuffer`, 40B
- `convertUnexpected`, 76B
- `countChar`, 99B
- `Counted::`
  - `~Counted`, 142B
  - `Counted`, 142B
  - `init`, 142B
  - `objectCount`, 142B
- `DataCollection::`
  - `avg`, 94B
  - `max`, 94B
  - `min`, 94B
- `DBPtr<T>::`
  - `DBPtr`, 160B, 161B
  - `operator T*`, 171B
- `deleteArray`, 18B
- `displayAndPlay`, 174B, 177B, 178B
- `displayInfo`, 49B, 50B
- `doSomething`, 69B, 71B, 72B

`drawLine`, 271B, 272B, 273B

`DynArray::operator[]`, 97B

`editTuple`, 161B, 167B

`EquipmentPiece::`
  - `EquipmentPiece`, 19B

`f`, 3B, 66B

`f1`, 61B, 73B

`f2`, 61B, 73B

`f3`, 61B

`f4`, 61B

`f5`, 61B

`find`, 281B, 282B, 283B

`findCubicleNumber`, 95B

`freeShared`, 42B

`FSA::`
  - `FSA`, 137B
  - `makeFSA`, 137B

`GameObject::collide`, 230B, 233B, 235B, 242B

`Graphic::`
  - `clone`, 126B
  - `operator<<`, 128B
  - `print`, 129B

`HeapTracked::`
  - `~HeapTracked`, 154B, 155B
  - `isOnHeap`, 154B, 155B
  - `operator delete`, 154B, 155B
  - `operator new`, 154B, 155B

`Image::Image`, 50B

`initializeCollisionMap`, 245B, 246B

`inventoryAsset`, 156B

`isSafeToDelete`, 153B

`Kitten::processAdoption`, 46B
`LargeObject::`
  - `field1`, 87B, 88B, 89B, 90B
  - `field2`, 87B, 88B
  - `field3`, 87B, 88B
  - `field4`, 87B, 88B
  - `field5`, 87B
  - `LargeObject`, 87B, 88B, 89B

`Lizard::operator=`, 259B, 260B, 261B, 262B, 263B, 264B, 265B
`LogEntry::`
  - `~LogEntry`, 161B
  - `LogEntry`, 161B

`lookup`, 245B, 247B
`main`, 111B, 251B, 274B
`makeStringPair`, 245B, 246B
`mallocShared`, 42B
`merge`, 172B
`MusicProduct::`
  - `displayTitle`, 173B
  - `MusicProduct`, 173B
  - `play`, 173B

`Name::Name`, 25B
`NewsLetter::`
  - `NewsLetter`, 125B, 127B
  - `readComponent`, 125B

`NLComponent::`
  - `clone`, 126B
  - `operator<<`, 128B
  - `print`, 129B

`normalize`, 170B
`onHeap`, 150B

`operator delete`, 41B, 153B

`operator new`, 38B, 40B, 153B

`operator*`, 102B, 103B, 104B

`operator+`, 100B, 105B, 106B, 107B, 108B, 109B

`operator-`, 107B, 108B

`operator<<`, 129B

`operator=`, 6B

`operator==`, 27B, 31B, 73B

`operator>>`, 62B

`passAndThrowWidget`, 62B, 63B

`printBSTArray`, 17B

`printDouble`, 10B

`Printer::`

- `~Printer`, 135B, 138B, 143B
- `makePrinter`, 138B, 139B, 140B, 143B
- `performSelfTest`, 130B, 139B, 143B
- `Printer`, 131B, 132B, 135B, 139B, 140B, 143B
- `reset`, 130B, 139B, 143B
- `submitJob`, 130B, 139B, 143B
- `thePrinter`, 132B

`PrintingStuff::Printer::`

- `performSelfTest`, 132B
- `Printer`, 133B
- `reset`, 132B
- `submitJob`, 132B

`PrintingStuff::thePrinter`, 132B, 133B

`PrintJob::PrintJob`, 131B

`printTreeNode`, 164B, 165B

`processAdoptions`, 46B, 47B, 48B

`processCollision`, 245B

`processInput`, 213B, 214B

`processTuple`, 171B

`Puppy::processAdoption`, 46B

`rangeCheck`, 35B

`Rational::`

- `asDouble`, 26B
- `denominator`, 102B, 225B
- `numerator`, 102B, 225B
- `operator double`, 25B
- `operator+=`, 107B
- `operator-=`, 107B
- `Rational`, 25B, 102B, 225B

`RCIPtr::`

- `~RCIPtr`, 209B
- `CountHolder`, 209B
- `init`, 209B
- `operator*`, 209B, 210B
- `operator=`, 209B, 210B
- `operator->`, 209B, 210B
- `RCIPtr`, 209B

`RCIPtr::CountHolder::`

- `~CountHolder`, 209B

`RCObject::`

- `~RCObject`, 194B, 204B, 205B
- `addReference`, 195B, 204B, 205B
- `isShareable`, 195B, 204B, 205B
- `isShared`, 195B, 204B, 205B
- `markUnshareable`, 195B, 204B, 205B
- `operator=`, 194B, 195B, 204B, 205B
- `RCObject`, 194B, 195B, 204B, 205B
- `removeReference`, 195B, 204B, 205B

`RCPtr::`

- `~RCPtr`, 199B, 202B, 203B, 206B
- `init`, 199B, 200B, 203B, 206B
- `operator*`, 199B, 203B, 206B
- `operator=`, 199B, 202B, 203B, 206B
- `operator->`, 199B, 203B, 206B
- `RCPtr`, 199B, 203B, 206B

`RCWidget::`

- `doThis`, 210B
- `RCWidget`, 210B
- `showThat`, 210B

`realMain`, 274B

`RegisterCollisionFunction::`

- `RegisterCollisionFunction`, 250B

`restoreAndProcessObject`, 88B

`reverse`, 36B

`satelliteAsteroid`, 251B

`satelliteShip`, 251B

`Session::`

- `~Session`, 59B, 60B, 61B, 77B
- `logCreation`, 59B
- `logDestruction`, 59B, 77B
- `Session`, 59B, 61B

`shipAsteroid`, 245B, 248B, 250B

`shipStation`, 245B, 250B

`simulate`, 272B, 273B

`SmartPtr<Cassette>::`

- `operator SmartPtr<MusicProduct>`, 175B

`SmartPtr<CD>::`

- `operator SmartPtr<MusicProduct>`, 175B

`SmartPtr<T>::`

    ~SmartPtr, 160B, 166B
    operator SmartPtr<U>, 176B
    operator void*, 168B
    operator!, 169B
    operator*, 160B, 166B, 176B
    operator=, 160B
    operator->, 160B, 167B, 176B
    SmartPtr, 160B, 176B
someFunction, 68B, 69B, 71B
SpaceShip::
    collide, 230B, 231B, 233B, 234B, 235B, 237B, 243B
    hitAsteroid, 235B, 236B, 243B, 244B
    hitSpaceShip, 235B, 236B, 243B
    hitSpaceStation, 235B, 236B, 243B
initializeCollisionMap, 239B, 240B, 241B, 243B
    lookup, 236B, 238B, 239B, 240B
SpecialWindow::
    height, 269B
    repaint, 269B
    resize, 269B
    width, 269B
stationAsteroid, 245B
stationShip, 245B
String::
    ~String, 188B
    markUnshareable, 207B
    operator=, 183B, 184B, 189B
    operator[], 190B, 191B, 194B, 204B, 207B, 218B, 219B, 220B, 221B
    String, 183B, 187B, 188B, 193B, 204B, 207B
String::CharProxy::
    CharProxy, 219B, 222B

    `operator char`, 219B, 222B
    `operator&`, 224B
    `operator=`, 219B, 222B, 223B
`String::StringValue::`
    `~StringValue`, 186B, 193B, 197B, 204B, 207B
    `init`, 204B, 206B
    `StringValue`, 186B, 193B, 197B, 201B, 204B, 206B, 207B
`swap`, 99B, 226B
`testBookEntryClass`, 52B, 53B
`TextBlock::`
    `clone`, 126B
    `operator<<`, 128B
    `print`, 129B
`thePrinter`, 130B, 131B, 134B
`Tuple::`
    `displayEditDialog`, 161B
    `isValid`, 161B
`TupleAccessors::`
    `TupleAccessors`, 172B
`TVStation::TVStation`, 226B
`twiddleBits`, 272B, 273B
`update`, 13B
`updateViaRef`, 14B
`UPInt::`
    `operator--`, 32B
    `operator++`, 32B, 33B
    `operator+=`, 32B
    `UPInt`, 105B
`UPNumber::`
    `~UPNumber`, 146B
    `destroy`, 146B

- `operator delete`, 157B
- `operator new`, 148B, 157B
- `UPNumber`, 146B, 148B

`uppercasify`, 100B

`Validation_error::what`, 70B

`watchTV`, 227B

`Widget::`

- `~Widget`, 210B
- `doThis`, 210B
- `operator=`, 210B
- `showThat`, 210B
- `Widget`, 40B, 210B

`Window::`

- `height`, 269B
- `repaint`, 269B
- `resize`, 269B
- `width`, 269B

`WindowHandle::`

- `~WindowHandle`, 49B
- `operator WINDOW_HANDLE`, 49B
- `operator=`, 49B
- `WindowHandle`, 49B

Footnotes

Basics

- † A second, unrelated use of `dynamic_cast` is to find the beginning of the memory occupied by an object. We explore that capability in Item 27.

Operators

- † Alas, it is not uncommon for compilers to fail to enforce this restriction. Before you write programs that rely on it, test your compilers to make sure they behave correctly.

Exceptions

- † A complete implementation of an almost-standard `auto_ptr` appears on pages 291-294.
- † Well, almost always. If the exception is not caught, the program will terminate. In that case, there is no guarantee that local objects (such as `w` in the example) will have their destructors called. Some compilers call them, some do not. Both behaviors are valid.
- † Provided, again, that the exception is caught.
- † Now there is. In July 1995, the ISO/ANSI standardization committee for C++ added a function, `uncaught_exception`, that returns `true` if an exception is active and has not yet been caught.
- † Compiler writers are actually allowed a slight bit of leeway regarding the "mandatory" nature of the copying; it can be eliminated under certain circumstances. Similar leeway provides the foundation for the return value optimization (see Item 20).
- † Alas, it can't, at least not portably. Though many compilers accept the code shown on this page, the standardization committee has inexplicably decreed that "an exception specification shall not appear in a typedef." I

don't know why. If you need a portable solution, you must — it hurts me to write this — make `CallBackPtr` a macro, sigh.

Efficiency

† In July 1995, the ISO/ANSI committee standardizing C++ added a requirement that most STL iterators support the "`->`" operator, so `it->second` should now work. Some STL implementations fail to satisfy this requirement, however, so `(*it).second` is still the more portable construct.

† In July 1996, the ISO/ANSI standardization committee declared that both named and unnamed objects may be optimized away via the return value optimization.

† At least that's what's supposed to happen. Alas, some compilers treat `T(lhs)` as a *cast* to remove `lhs`'s `const`ness, then add `rhs` to `lhs` and return a reference to the modified `lhs`! Test your compilers before relying on the behavior described above.

Techniques

† In July 1996, the ISO/ANSI standardization committee changed the default linkage of inline functions to *external*, so the problem I describe here has been eliminated, at least on paper. Your compilers may not yet be in accord with the standard, however, so your best bet is still to shy away from inline functions with static data.

† I have since become convinced that signature-based techniques are all but foolproof. For details, consult `http://www.aristeia.com/BookErrata/M27Comments.html`.

† The `string` type in the standard C++ library (see Item 35) uses a combination of solutions two and three. The reference returned from the non-`const` `operator[]` is guaranteed to be valid until the next function call that might modify the string. After that, use of the reference (or the character to which it refers) yields undefined results. This allows the string's shareability flag to be reset to `true` whenever a function is called that might modify the string.

[†] It turns out that it's not so predictable after all. The C++ standard doesn't specify the return value of `type_info::name`, and different implementations do behave differently. A preferable design is to use a container-friendly class that wraps `type_info` objects, such as Andrei Alexandrescu's `TypeInfo` class, which is described in section 2.8 of his *Modern C++ Design* (Addison Wesley, 2001).

Recommended Reading

[†] As of January 2008, both of these magazines have ceased publication. There are currently no broad-circulation paper publications devoted to C++.

An `auto_ptr` Implementation

[†] This is primarily because the specification for `auto_ptr` has for years been a moving target. The final specification was adopted only in November 1997. For details, consult the `auto_ptr` information at this book's WWW and FTP sites (see page 8B). Note that the `auto_ptr` described here omits a few details present in the official version, such as the fact that `auto_ptr` is in the `std` namespace (see Item 35) and that its member functions promise not to throw exceptions.

Effective STL

50 Specific Ways to Improve Your Use of the Standard Template Library

Scott Meyers

ADDISON–WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and we were aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The excerpt from *How the Grinch Stole Christmas!* by Dr. Seuss is trademarked and copyright © Dr. Seuss Enterprises, L.P., 1957 (renewed 1985). Used by permission of Random House Children's Books, a division of Random House, Inc.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact:

Pearson Education Corporate Sales Division

201 W. 103rd Street

Indianapolis, IN 46290

(800) 428-5331

corpsales@pearsoned.com

Visit AW on the Web: www.awl.com/cseng/

Library of Congress Cataloging-in-Publication Data

Meyers, Scott (Scott Douglas)

Effective STL: 50 specific ways to improve your use of the standard template library/Scott Meyers.

p. cm.— (Addison-Wesley professional computing series)

Includes bibliographical references and index.

ISBN: 0-201-74962-9

1. C++ (Computer program language) 2. Standard template library. I. Title. II. Series.

QA76.73.C153.M49 2001

005.13'3—dc21

2001022851

CIP

Copyright © 2001 by Addison-Wesley

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

ISBN 0201749629

Text printed in the United States on recycled paper at Courier Westford in Westford, Massachusetts.

10th Printing September 2007

For Woofieland.

Preface

> *It came without ribbons! It came without tags!*
> *It came without packages, boxes or bags!*
>
> — Dr. Seuss, *How the Grinch Stole Christmas!*, Random House, 1957

I first wrote about the Standard Template Library in 1995, when I concluded the final Item of *More Effective C++* with a brief STL overview. I should have known better. Shortly thereafter, I began receiving mail asking when I'd write *Effective STL*.

I resisted the idea for several years. At first, I wasn't familiar enough with the STL to offer advice on it, but as time went on and my experience with it grew, this concern gave way to other reservations. There was never any question that the library represented a breakthrough in efficient and extensible design, but when it came to *using* the STL, there were practical problems I couldn't overlook. Porting all but the simplest STL programs was a challenge, not only because library implementations varied, but also because template support in the underlying compilers ranged from good to awful. STL tutorials were hard to come by, so learning "the STL way of programming" was difficult, and once that hurdle was overcome, finding comprehensible and accurate reference documentation was a challenge. Perhaps most daunting, even the smallest STL usage error often led to a blizzard of compiler diagnostics, each thousands of characters long, most referring to classes, functions, or templates not mentioned in the offending source code, almost all incomprehensible. Though I had great admiration for the STL and for the people behind it, I felt uncomfortable recommending it to practicing programmers. I wasn't sure it was *possible* to use the STL effectively.

Then I began to notice something that took me by surprise. Despite the portability problems, despite the dismal documentation, despite the compiler diagnostics resembling transmission line noise, many of my consulting clients were using the STL anyway. Furthermore, they weren't just playing with it, they were using it in production code! That was a revelation. I knew that the STL featured an elegant design, but any library for which programmers are willing to endure portability headaches, poor documentation, and incomprehensible error messages has a lot more going for it than just good design. For an increasingly large number of professional programmers, I realized, even a bad implementation of the STL was preferable to no implementation at all.

Furthermore, I knew that the situation regarding the STL would only get better. Libraries and compilers would grow more conformant with the Standard (they have), better documentation would become available (it has—consult the bibliography beginning on page 225), and compiler diagnostics would improve (for the most part, we're still waiting, but Item 49 offers suggestions for how to cope while we wait). I therefore decided to chip in and do my part for the STL movement. This book is the result: 50 specific ways to improve your use of C++'s Standard Template Library.

My original plan was to write the book in the second half of 1999, and with that thought in mind, I put together an outline. But then I changed course. I suspended work on the book and developed an introductory training course on the STL, which I then taught several times to groups of programmers. About a year later, I returned to the book, significantly revising the outline based on my experiences with the training course. In the same way that my *Effective C++* has been successful by being grounded in the problems faced by real programmers, it's my hope that *Effective STL* similarly addresses the practical aspects of STL programming—the aspects most important to professional developers.

I am always on the lookout for ways to improve my understanding of C++. If you have suggestions for new guidelines for STL programming or if you have comments on the guidelines in this book, please let me know. In addition, it is my continuing goal to make this book as accurate as possible, so for each error in this book that is reported to me—be it technical, grammatical, typographical, or otherwise—I will, in future printings, gladly add to the acknowledgments the name of the first person to bring that error to my attention. Send your suggested guidelines, your comments, and your criticisms to estl@aristeia.com.

I maintain a list of changes to this book since its first printing, including bug-fixes, clarifications, and technical updates. The list is available at the *Effective STL Errata* web site, http://www.aristeia.com/BookErrata/estl1e-errata.html.

If you'd like to be notified when I make changes to this book, I encourage you to join my mailing list. I use the list to make announcements likely to be of interest to people who follow my work on C++. For details, consult http://www.aristeia.com/MailingList/.

SCOTT DOUGLAS MEYERS STAFFORD, OREGON
http://www.aristeia.com/ APRIL
2001

Acknowledgments

I had an enormous amount of help during the roughly two years it took me to make some sense of the STL, create a training course on it, and write this book. Of all my sources of assistance, two were particularly important. The first is Mark Rodgers. Mark generously volunteered to review my training materials as I created them, and I learned more about the STL from him than from anybody else. He also acted as a technical reviewer for this book, again providing observations and insights that improved virtually every Item.

The other outstanding source of information was several C++-related Usenet newsgroups, especially `comp.lang.c++.moderated` (“`clcm`”), `comp.std.c++`, and `microsoft.public.vc.stl`. For well over a decade, I’ve depended on the participants in newsgroups like these to answer my questions and challenge my thinking, and it’s difficult to imagine what I’d do without them. I am deeply grateful to the Usenet community for their help with both this book and my prior publications on C++.

My understanding of the STL was shaped by a variety of publications, the most important of which are listed in the Bibliography. I leaned especially heavily on Josuttis’ *The C++ Standard Library* [3].

This book is fundamentally a summary of insights and observations made by others, though a few of the ideas are my own. I’ve tried to keep track of where I learned what, but the task is hopeless, because a typical Item contains information garnered from many sources over a long period of time. What follows is incomplete, but it’s the best I can do. Please note that my goal here is to summarize where *I* first learned of an idea or technique, not where the idea or technique was originally developed or who came up with it.

In Item 1, my observation that node-based containers offer better support for transactional semantics is based on section 5.11.2 of Josuttis’ *The C++ Standard Library* [3]. Item 2 includes an example from Mark Rodgers on how `typedef`s help when allocator types are changed. Item 5 was motivated by Reeves’ *C++ Report* column, “STL Gotchas” [17]. Item 8 sprang from Item 37 in Sutter’s *Exceptional C++* [8], and Kevlin Henney provided important details on how containers of `auto_ptr`s fail in practice. In Usenet postings, Matt Austern provided examples of when allocators are useful, and I include his examples in Item 11. Item 12 is based on the discussion of thread safety at the SGI STL web site [21]. The material in Item 13 on the performance implications of reference counting in a multithreaded environment is drawn from Sutter's writings on this topic [20]. The idea for Item 15 came from Reeves' *C++ Report* column, "Using Standard `string` in the Real World, Part 2," [18]. In Item 16, Mark Rodgers came up with the technique I show for having a C API write data directly into a `vector`. Item 17 includes information from Usenet postings by Siemel Naran and Carl Barron. I stole Item 18 from Sutter's *C++ Report* column, "When Is a Container Not a Container?" [12]. In Item 20, Mark Rodgers contributed the idea of transforming a pointer into an object via a dereferencing functor, and Scott Lewandowski came up with the version of `DereferenceLess` I present. Item 21 originated in a Doug Harrison posting to `microsoft.public.vc.stl`, but the decision to restrict the focus of that Item to equality was mine. I based Item 22 on Sutter's *C++ Report* column, "Standard Library News: `set`s and `map`s" [13]; Matt Austern helped me understand the status of the Standardization Committee's Library Issue #103. Item 23 was inspired by Austern's *C++ Report* article, "Why You Shouldn't Use `set`—and What to Use Instead" [15]; David Smallberg provided a neat refinement for my implementation of `DataCompare`. My description of Dinkumware's hashed containers is based on Plauger's *C/C++ Users Journal* column, "Hash Tables" [16]. Mark Rodgers doesn't agree with the overall advice of Item 26, but an early motivation for that Item was his observation that some container member functions accept only arguments of type `iterator`. My treatment of Item 29 was motivated and informed by Usenet discussions involving Matt Austern and James Kanze; I was also influenced by Kreft and Langer's *C++ Report* article, "A Sophisticated Implementation of User-Defined Inserters and Extractors" [25]. Item 30 is due to a discussion in section 5.4.2 of Josuttis' *The C++ Standard Library* [3]. In Item 31, Marco Dalla Gasperina contributed the example use of `nth_element` to calculate medians, and use of that algorithm for finding percentiles comes straight out of section 18.7.1 of Stroustrup's *The C++ Programming Language* [7]. Item 32 was influenced by the material in section 5.6.1 of Josuttis' *The C++ Standard Library* [3]. Item 35 originated in Austern's *C++ Report* column "How to Do Case-Insensitive String Comparison" [11], and James Kanze's and John Potter's `clcm` postings helped me refine my understanding of the issues involved. Stroustrup's implementation for `copy_if`, which I show in Item 36, is from section 18.6.1 of his *The C++ Programming Language* [7].

Item 39 was largely motivated by the publications of Josuttis, who has written about “stateful predicates” in his *The C++ Standard Library* [3], in Standard Library Issue #92, and in his *C++ Report* article, “Predicates vs. Function Objects” [14]. In my treatment, I use his example and recommend a solution he has proposed, though the use of the term “pure function” is my own. Matt Austern confirmed my suspicion in Item 41 about the history of the terms `mem_fun` and `mem_fun_ref`. Item 42 can be traced to a lecture I got from Mark Rodgers when I considered violating that guideline. Mark Rodgers is also responsible for the insight in Item 44 that non-member searches over `map`s and `multimap`s examine both components of each pair, while member searches examine only the first (key) component. Item 45 contains information from various `clcm` contributors, including John Potter, Marcin Kasperski, Pete Becker, Dennis Yelle, and David Abrahams. David Smallberg alerted me to the utility of `equal_range` in performing equivalence-based searches and counts over sorted sequence containers. Andrei Alexandrescu helped me understand the conditions under which “the reference-to-reference problem” I describe in Item 50 arises, and I modeled my example of the problem on a similar example provided by Mark Rodgers at the Boost Web Site [22].

Credit for the material in Appendix A goes to Matt Austern, of course. I’m grateful that he not only gave me permission to include it in this book, he also tweaked it to make it even better than the original.

Good technical books require a thorough pre-publication vetting, and I was fortunate to benefit from the insights of an unusually talented group of technical reviewers. Brian Kernighan and Cliff Green offered early comments on a partial draft, and complete versions of the manuscript were scrutinized by Doug Harrison, Brian Kernighan, Tim Johnson, Francis Glassborow, Andrei Alexandrescu, David Smallberg, Aaron Campbell, Jared Manning, Herb Sutter, Stephen Dewhurst, Matt Austern, Gillmer Derge, Aaron Moore, Thomas Becker, Victor Von, and, of course, Mark Rodgers. Katrina Avery did the copyediting.

One of the most challenging parts of preparing a book is finding good technical reviewers. I thank John Potter for introducing me to Jared Manning and Aaron Campbell.

Herb Sutter kindly agreed to act as my surrogate in compiling, running, and reporting on the behavior of some STL test programs under a beta version of

Microsoft’s Visual Studio .NET, while Leor Zolman undertook the herculean task of testing all the code in this book. Any errors that remain are my fault, of course, not Herb’s or Leor’s.

Angelika Langer opened my eyes to the indeterminate status of some aspects of STL function objects. This book has less to say about function objects than it otherwise might, but what it does say is more likely to remain true. At least I hope it is.

This printing of the book is better than earlier printings, because I was able to address problems identified by the following sharp-eyed readers: Jon Webb, Michael Hawkins, Derek Price, Jim Scheller, Carl Manaster, Herb Sutter, Albert Franklin, George King, Dave Miller, Harold Howe, John Fuller, Tim McCarthy, John Hershberger, Igor Mikolic-Torreira, Stephan Bergmann, Robert Allan Schwartz, John Potter, David Grigsby, Sanjay Pattni, Jesper Andersen, Jing Tao Wang, André Blavier, Dan Schmidt, Bradley White, Adam Petersen, Wayne Goertel, Gabriel Netterdag, Jason Kenny, Scott Blachowicz, Seyed H. Haeri, Gareth McCaughan, Giulio Agostini, Fraser Ross, Wolfram Burkhardt, Keith Stanley, Leor Zolman, Chan Ki Lok, Motti Abramsky, Kevlin Henney, Stefan Kuhlins, Phillip Ngan, Jim Phillips, Ruediger Dreier, Guru Chandar, Charles Brockman, Day Barr, Eric Niebler, Sharad Kala, Declan Moran, Nick de Smith, David Callaway, Shlomi Frank, Andrea Griffini, Hans Eckardt, David Smallberg, Matt Page, Andy Fyfe, Vincent Stojanov, Randy Parker, Thomas Schell, Cameron Mac Minn, Mark Davis, and Giora Unger. I’m grateful for their help in improving *Effective STL*.

My collaborators at Addison-Wesley included John Wait (my editor and now a senior VP), Alicia Carey and Susannah Buzard (his assistants n and n+1), John Fuller (the production coordinator), Karin Hansen (the cover designer), Jason Jones (all-around technical guru, especially with respect to the demonic software spewed forth by Adobe), Marty Rabinowitz (their boss, but he works, too), and Curt Johnson, Chanda Leary-Coutu, and Robin Bruce (all marketing people, but still very nice).

Abbi Staley made Sunday lunches a routinely pleasurable experience.

As she has for the six books and one CD that came before it, my wife, Nancy, tolerated the demands of my research and writing with her usual forbearance and offered me encouragement and support when I needed it most. She never fails to remind me that there’s more to life than C++ and software.

And then there’s our dog, Persephone. As I write this, it is her sixth birthday. Tonight, she and Nancy and I will visit Baskin-Robbins for ice cream. Persephone will have vanilla. One scoop. In a cup. To go.

Introduction

You’re already familiar with the STL. You know how to create containers, iterate over their contents, add and remove elements, and apply common algorithms, such as `find` and `sort`. But you’re not satisfied. You can’t shake the sensation that the STL offers more than you’re taking advantage of. Tasks that should be simple aren’t. Operations that should be straightforward leak resources or behave erratically. Procedures that should be efficient demand more time or memory than you’re willing to give them. Yes, you know how to use the STL, but you’re not sure you’re using it *effectively*.

I wrote this book for you.

In *Effective STL,* I explain how to combine STL components to take full advantage of the library’s design. Such information allows you to develop simple, straightforward solutions to simple, straightforward problems, and it also helps you design elegant approaches to more complicated problems. I describe common STL usage errors, and I show you how to avoid them. That helps you dodge resource leaks, code that won’t port, and behavior that is undefined. I discuss ways to optimize your code, so you can make the STL perform like the fast, sleek machine it is intended to be.

The information in this book will make you a better STL programmer. It will make you a more productive programmer. And it will make you a happier programmer. Using the STL is fun, but using it effectively is outrageous fun, the kind of fun where they have to drag you away from the keyboard, because you just can’t believe the good time you’re having. Even a cursory glance at the STL reveals that it is a wondrously cool library, but the coolness runs broader and deeper than you probably imagine. One of my primary goals in this book is to convey to you just how amazing the library is, because in the nearly 30 years I’ve been programming, I’ve never seen anything like the STL. You probably haven’t either.

Defining, Using, and Extending the STL

There is no official definition of “the STL,” and different people mean different things when they use the term. In this book, “the STL” means the parts of C++’s Standard Library that work with iterators. That includes the standard containers (including `string`), parts of the iostream library, function objects, and algorithms. It excludes the standard container adapters (`stack`, `queue`, and `priority_queue`) as well as the containers `bitset` and `valarray`, because they lack iterator support. It doesn't include arrays, either. True, arrays support iterators in the form of pointers, but arrays are part of the C++ *language*, not the library.

Technically, my definition of the STL excludes extensions of the standard C++ library, notably hashed containers, singly linked lists, ropes, and a variety of nonstandard function objects. Even so, an effective STL programmer needs to be aware of such extensions, so I mention them where it's appropriate. Indeed, Item 25 is devoted to an overview of nonstandard hashed containers. They're not in the STL now, but something similar to them is almost certain to make it into the next version of the standard C++ library, and there's value in glimpsing the future.

One of the reasons for the existence of STL extensions is that the STL is a library designed to be extended. In this book, however, I focus on *using* the STL, not on adding new components to it. You'll find, for example, that I have little to say about writing your own algorithms, and I offer no guidance at all on writing new containers and iterators. I believe that it's important to master what the STL already provides before you embark on increasing its capabilities, so that's what I focus on in *Effective STL*. When you decide to create your own STLesque components, you'll find advice on how to do it in books like Josuttis' *The C++ Standard Library* [3] and Austern's *Generic Programming and the STL* [4]. One aspect of STL extension I *do* discuss in this book is writing your own function objects. You can't use the STL effectively without knowing how to do that, so I've devoted an entire chapter to the topic (Chapter 6).

Citations

The references to the books by Josuttis and Austern in the preceding paragraph demonstrate how I handle bibliographic citations. In general, I try to mention enough of a cited work to identify it for people who are already familiar with it. If you already know about these authors' books, for example, you don't have to turn to the Bibliography to find out that [3] and [4] refer to books you already know. If you're not familiar with a publication, of course, the Bibliography (which begins on page 225) gives you a full citation.

I cite three works often enough that I generally leave off the citation number. The first of these is the International Standard for C++ [5], which I usually refer to as simply "the Standard." The other two are my earlier books on C++, *Effective C++* [1] and *More Effective C++* [2].

The STL and Standards

I refer to the C++ Standard frequently, because *Effective STL* focuses on portable, standard-conformant C++. In theory, everything I show in this book will work with every C++ implementation. In practice, that isn't true. Shortcomings in compiler and STL implementations conspire to prevent some valid code from compiling or from behaving the way it's supposed to. Where that is commonly the case, I describe the problems, and I explain how you can work around them.

Sometimes, the easiest workaround is to use a different STL implementation. Appendix B gives an example of when this is the case. In fact, the more you work with the STL, the more important it becomes to distinguish between your *compilers* and your *library implementations*. When programmers run into difficulties trying to get legitimate code to compile, it's customary for them to blame their compilers, but with the STL, compilers can be fine, while STL implementations are faulty. To emphasize the fact that you are dependent on both your compilers and your library implementations, I refer to your *STL platforms*. An STL platform is the combination of a particular compiler and a particular STL implementation. In this book, if I mention a compiler problem, you can be sure that I mean it's the compiler that's the culprit. However, if I refer to a problem with your STL platform, you should interpret that as "maybe a compiler bug, maybe a library bug, possibly both."

I generally refer to your "compilers"—*plural*. That's an outgrowth of my longstanding belief that you improve the quality (especially the portability) of your code if you ensure that it works with more than one compiler. Furthermore, using multiple compilers generally makes it easier to unravel the Gordian nature of error messages arising from improper use of the STL. (Item 49 is devoted to approaches to decoding such messages.)

Another aspect of my emphasis on standard-conforming code is my concern that you avoid constructs with undefined behavior. Such constructs may do anything at runtime. Unfortunately, this means they may do precisely what you want them to, and that can lead to a false sense of security. Too many programmers assume that undefined behavior always leads to an obvious problem, e.g., a segmentation fault or other catastrophic failure. The results of undefined behavior can actually be much more subtle, e.g., corruption of rarely-referenced data. They can also vary across program runs. I find that a good working definition of undefined behavior is “works for me, works for you, works during development and QA, but blows up in your most important customer’s face.” It’s important to avoid undefined behavior, so I point out common situations where it can arise. You should train yourself to be alert for such situations.

Reference Counting

It’s close to impossible to discuss the STL without mentioning reference counting. As you’ll see in Items 7 and 33, designs based on containers of pointers almost invariably lead to reference counting. In addition, many `string` implementations are internally reference counted, and, as Item 15 explains, this may be an implementation detail you can’t afford to ignore. In this book, I assume that you are familiar with the basics of reference counting. If you’re not, most intermediate and advanced C++ texts cover the topic. In *More Effective C++*, for example, the relevant material is in Items 28 and 29. If you don’t know what reference counting is and you have no inclination to learn, don’t worry. You’ll get through this book just fine, though there may be a few sentences here and there that won’t make as much sense as they otherwise would.

`string` and `wstring`

Whatever I say about `string` applies equally well to its wide-character counterpart, `wstring`. Similarly, any time I refer to the relationship between `string` and `char` or `char*`, the same is true of the relationship between `wstring` and `wchar_t` or `wchar_t*`. In other words, just because I don’t explicitly mention wide-character strings in this book, don’t assume that the STL fails to support them. It supports them as well as `char`-based strings. It has to. Both `string` and `wstring` are instantiations of the same template, `basic_string`.

Terms, Terms, Terms

This is not an introductory book on the STL, so I assume you know the fundamentals. Still, the following terms are sufficiently important that I feel compelled to review them:

- `vector`, `string`, `deque`, and `list` are known as the *standard sequence containers*. The *standard associative containers* are `set`, `multiset`, `map`, and `multimap`.
- Iterators are divided into five categories, based on the operations they support. Very briefly, *input iterators* are read-only iterators where each iterated location may be read only once. *Output iterators* are write-only iterators where each iterated location may be written only once. Input and output iterators are modeled on reading and writing input and output streams (e.g., files). It's thus unsurprising that the most common manifestations of input and output iterators are `istream_iterator`s and `ostream_iterator`s, respectively.

  Forward iterators have the capabilities of both input and output iterators, but they can read or write a single location repeatedly. They don't support `operator--`, so they can move only forward with any degree of efficiency. All standard STL containers support iterators that are more powerful than forward iterators, but, as you'll see in Item 25, one design for hashed containers yields forward iterators. Containers for singly linked lists (considered in Item 50) also offer forward iterators.

  Bidirectional iterators are just like forward iterators, except they can go backward as easily as they go forward. The standard associative containers all offer bidirectional iterators. So does `list`.

  Random access iterators do everything bidirectional iterators do, but they also offer "iterator arithmetic," i.e., the ability to jump forward or backward in a single step. `vector`, `string`, and `deque` each provide random access iterators. Pointers into arrays act as random access iterators for the arrays.
- Any class that overloads the function call operator (i.e., `operator()`) is a *functor class*. Objects created from such classes are known as *function objects* or *functors*. Most places in the STL that work with function objects work equally well with real functions, so I often use the term "function objects" to mean both C++ functions as well as true function objects.

- The functions `bind1st` and `bind2nd` are known as *binders*.

A revolutionary aspect of the STL is its complexity guarantees. These guarantees bound the amount of work any STL operation is allowed to perform. This is wonderful, because it can help you determine the relative efficiency of different approaches to the same problem, regardless of the STL platform you're using. Unfortunately, the terminology behind the complexity guarantees can be confusing if you haven't been formally introduced to the jargon of computer science. Here's a quick primer on the complexity terms I use in this book. Each refers to how long it takes to do something as a function of *n*, the number of elements in a container or range.

- An operation that runs in *constant time* has performance that is unaffected by changes in *n*. For example, inserting an element into a `list` is a constant-time operation. Regardless of whether the list has one element or one million, the insertion takes about the same amount of time.

  Don't take the term "constant time" too literally. It doesn't mean that the amount of time it takes to do something is literally constant, it just means that it's unaffected by *n*. For example, two STL platforms might take dramatically different amounts of time to perform the same "constant-time" operation. This could happen if one library has a much more sophisticated implementation than another or if one compiler performs substantially more aggressive optimization.

  A variant of constant time complexity is *amortized constant time*. Operations that run in amortized constant time are usually constant-time operations, but occasionally they take time that depends on *n*. Amortized constant time operations *typically* run in constant time.

- An operation that runs in *logarithmic time* needs more time to run as *n* gets larger, but the time it requires grows at a rate proportional to the logarithm of *n*. For example, an operation on a million items would be expected to take only about three times as long as on a hundred items, because $\log n^3 = 3 \log n$. Most search operations on associative containers (e.g., `set::find`) are logarithmic-time operations.
- The time needed to perform an operation that runs in *linear time* increases at a rate proportional to increases in *n*. The standard algorithm `count` runs in linear time, because it has to look at every

> element of the range it's given. If the range triples in size, it has to do three times as much work, and we'd expect it to take about three times as long to do it.

As a general rule, a constant-time operation runs faster than one requiring logarithmic time, and a logarithmic-time operation runs faster than one whose performance is linear. This is always true when *n* gets big enough, but for relatively small values of *n*, it's sometimes possible for an operation with a worse theoretical complexity to outperform an operation with a better theoretical complexity. If you'd like to know more about STL complexity guarantees, turn to Josuttis' *The C++ Standard Library* [3].

As a final note on terminology, recall that each element in a `map` or `multimap` has two components. I generally call the first component the *key* and the second component the *value*. Given

```
map<string, double> m;
```

for example, the `string` is the key and the `double` is the value.

Code Examples

This book is filled with example code, and I explain each example when I introduce it. Still, it's worth knowing a few things in advance.

You can see from the `map` example above that I routinely omit `#include`s and ignore the fact that STL components are in namespace `std`. When defining the map `m`, I could have written this,

```
#include <map>
#include <string>

using std::map;
using std::string;

map<string, double> m;
```

but I prefer to save us both the noise.

When I declare a formal type parameter for a template, I use `typename` instead of `class`. That is, instead of writing this,

```
template<class T>
class Widget { ... };
```

I write this:

```
template<typename T>
class Widget { ... };
```

In this context, `class` and `typename` mean exactly the same thing, but I find that `typename` more clearly expresses what I usually want to say: that *any* type will do; `T` need not be a class. If you prefer to use `class` to declare type parameters, go right ahead. Whether to use `typename` or `class` in this context is purely a matter of style.

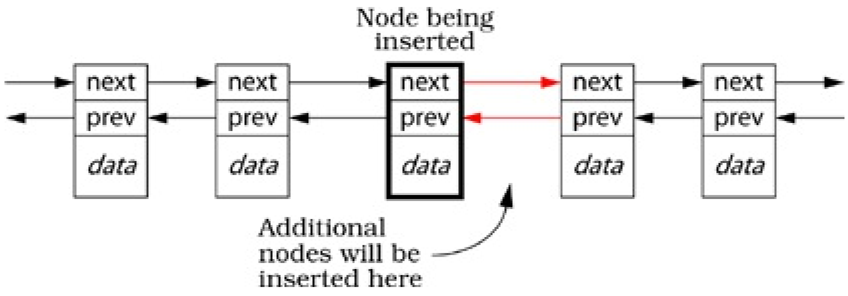
It is not a matter of style in a different context. To avoid potential parsing ambiguities (the details of which I'll spare you), you are required to use `typename` to precede type names that are dependent on formal type parameters. Such types are known as *dependent types*, and an example will help clarify what I'm talking about. Suppose you'd like to write a template for a function that, given an STL container, returns whether the last element in the container is greater than the first element. Here's one way to do it:

Click here to view code image

```
template<typename C>
bool lastGreaterThanFirst(const C& container)
{
  if (container.empty()) return false;

  typename C::const_iterator begin(container.begin());
  typename C::const_iterator end(container.end());

  return *--end > *begin;
}
```

In this example, the local variables `begin` and `end` are of type `C::const_iterator`. `const_iterator` is a type that is dependent on the formal type parameter `C`. Because `C::const_iterator` is a dependent type, you are required to precede it with the word `typename`. (Some compilers incorrectly accept the code without the `typename`s, but such code isn't portable.)

I hope you've noticed my use of color in the examples above. It's there to focus your attention on parts of the code that are particularly important. Often, I highlight the differences between related examples, such as when I showed the two possible ways to declare the parameter `T` in the `Widget` example. This use of color to call out especially noteworthy parts of examples carries over to diagrams, too. For instance, this diagram from Item 5 uses color to identify the two pointers that are affected when a new element is inserted into a `list`:



I also use color for chapter numbers, but such use is purely gratuitous. This being my first two-color book, I hope you'll forgive me a little chromatic exuberance.

Two of my favorite parameter names are `lhs` and `rhs`. They stand for "left-hand side" and "right-hand side," respectively, and I find them especially useful when declaring operators. Here's an example from Item 19:

Click here to view code image

```
class Widget { ... };
bool operator==(const Widget& lhs, const Widget& rhs);
```

When this function is called in a context like this,

```
if (x == y) ...                  // assume x and y are Widgets
```

`x`, which is on the left-hand side of the "`==`", is known as `lhs` inside `operator==`, and `y` is known as `rhs`.

As for the class name `Widget`, that has nothing to do with GUIs or toolkits. It's just the name I use for "some class that does something." Sometimes, as on page 7, `Widget` is a class template instead of a class. In such cases, you may find that I still refer to `Widget` as a class, even though it's really a template. Such sloppiness about the difference between classes and class templates, structs and struct templates, and functions and function templates hurts no one as long as there is no ambiguity about what is being discussed. In cases where it could be confusing, I do distinguish between templates and the classes, structs, and functions they generate.

Efficiency Items

I considered including a chapter on efficiency in *Effective STL*, but I ultimately decided that the current organization was preferable. Still, a number of Items focus on minimizing space and runtime demands. For your performance-enhancing convenience, here is the table of contents for the virtual chapter on efficiency:

Item 4: Call empty instead of checking size() against zero

Item 5: Prefer range member functions to their single-element counterparts

Item 14: Use reserve to avoid unnecessary reallocations

Item 15: Be aware of variations in string implementations

Item 23: Consider replacing associative containers with sorted vectors

Item 24: Choose carefully between map::operator[] and map::insert when efficiency is important

Item 25: Familiarize yourself with the nonstandard hashed containers

Item 29: Consider istreambuf_iterators for character-by-character input

Item 31: Know your sorting options

Item 44: Prefer member functions to algorithms with the same names

Item 46: Consider function objects instead of functions as algorithm parameters

The Guidelines in *Effective STL*

The guidelines that make up the 50 Items in this book are based on the insights and advice of the world's most experienced STL programmers. These guidelines summarize things you should almost always do—or almost always avoid doing—to get the most out of the Standard Template Library. At the same time, they're just guidelines. Under some conditions, it makes sense to violate them. For example, the title of Item 7 tells you to invoke `delete` on `new`ed pointers in a container before the container is destroyed, but the text of that Item makes clear that this applies only when the objects pointed to by those pointers should go away when the container does. This is often the case, but it's not universally true. Similarly, the title of Item 35 beseeches you to use STL algorithms to perform simple case-insensitive string comparisons, but the text of the Item points out that in some cases, you'll be better off using a function that's not only outside the STL, it's not even part of standard C++!

Only you know enough about the software you’re writing, the environment in which it will run, and the context in which it’s being created to determine whether it’s reasonable to violate the guidelines I present. Most of the time, it won’t be, and the discussions that accompany each Item explain why. In a few cases, it will. Slavish devotion to the guidelines isn’t appropriate, but neither is cavalier disregard. Before venturing off on your own, you should make sure you have a good reason.

Chapter 1. Containers

Sure, the STL has iterators, algorithms, and function objects, but for most C++ programmers, it's the containers that stand out. More powerful and flexible than arrays, they grow (and often shrink) dynamically, manage their own memory, keep track of how many objects they hold, bound the algorithmic complexity of the operations they support, and much, much more. Their popularity is easy to understand. They're simply better than their competition, regardless of whether that competition comes from containers in other libraries or is a container type you'd write yourself. STL containers aren't just good. They're *really* good.

This chapter is devoted to guidelines applicable to all the STL containers. Later chapters focus on specific container types. The topics addressed here include selecting the appropriate container given the constraints you face; avoiding the delusion that code written for one container type is likely to work with other container types; the significance of copying operations for objects in containers; difficulties that arise when pointers or `auto_ptr`s are stored in containers; the ins and outs of erasing; what you can and cannot accomplish with custom allocators; tips on how to maximize efficiency; and considerations for using containers in a threaded environment.

That's a lot of ground to cover, but don't worry. The Items break it down into bite-sized chunks, and along the way, you're almost sure to pick up several ideas you can apply to your code *now*.

Item 1: Choose your containers with care

You know that C++ puts a variety of containers at your disposal, but do you realize just how varied that variety is? To make sure you haven't overlooked any of your options, here's a quick review.

- The **standard STL sequence containers**, `vector`, `string`, `deque`, and `list`.
- The **standard STL associative containers**, `set`, `multiset`, `map`, and `multimap`.
- The **nonstandard sequence containers** `slist` and `rope`. `slist` is a singly linked list, and `rope` is essentially a heavy-duty `string`. (A "rope" is a

heavy-duty "string." Get it?) You'll find a brief overview of these nonstandard (but commonly available) containers in Item 50.

- The **nonstandard associative containers** `hash_set`, `hash_multiset`, `hash_map`, and `hash_multimap`. I examine these widely available hash-table-based variants on the standard associative containers in Item 25.
- **`vector<char>` as a replacement for `string`**. Item 13 describes the conditions under which such a replacement might make sense.
- **`vector` as a replacement for the standard associative containers**. As Item 23 makes clear, there are times when `vector` can outperform the standard associative containers in both time and space.
- Several **standard non-STL containers**, including arrays, `bitset`, `valarray`, `stack`, `queue`, and `priority_queue`. Because these are non-STL containers, I have little to say about them in this book, though Item 16 mentions a case where arrays are preferable to STL containers and Item 18 explains why `bitset` may be better than `vector<bool>`. It's also worth bearing in mind that arrays can be used with STL algorithms, because pointers can be used as array iterators.

That's a panoply of options, and it's matched in richness by the range of considerations that should go into choosing among them. Unfortunately, most discussions of the STL take a fairly narrow view of the world of containers, ignoring many issues relevant to selecting the one that is most appropriate. Even the Standard gets into this act, offering the following guidance for choosing among `vector`, `deque`, and `list`:

> `vector`, `list`, and `deque` offer the programmer different complexity trade-offs and should be used accordingly. `vector` is the type of sequence that should be used by default. `list` should be used when there are frequent insertions and deletions from the middle of the sequence. `deque` is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.

If your primary concern is algorithmic complexity, I suppose this constitutes reasonable advice, but there is so much more to be concerned with.

In a moment, we'll examine some of the important container-related issues that complement algorithmic complexity, but first I need to introduce a way of categorizing the STL containers that isn't discussed as often as it should be. That is the distinction between contiguous-memory containers and node-based containers.

Contiguous-memory containers (also known as *array-based containers*) store their elements in one or more (dynamically allocated) chunks of memory, each chunk holding more than one container element. If a new element is inserted or an existing element is erased, other elements in the same memory chunk have to be shifted up or down to make room for the new element or to fill the space formerly occupied by the erased element. This kind of movement affects both performance (see Items 5 and 14) and exception safety (as we'll soon see). The standard contiguous-memory containers are `vector`, `string`, and `deque`. The nonstandard `rope` is also a contiguous-memory container.

Node-based containers store only a single element per chunk of (dynamically allocated) memory. Insertion or erasure of a container element affects only pointers to nodes, not the contents of the nodes themselves, so element values need not be moved when something is inserted or erased. Containers representing linked lists, such as `list` and `slist`, are node-based, as are all the standard associative containers. (They're typically implemented as balanced trees.) The nonstandard hashed containers use varying node-based implementations, as you'll see in Item 25.

With this terminology out of the way, we're ready to sketch some of the questions most relevant when choosing among containers. In this discussion, I omit consideration of non-STL-like containers (e.g., arrays, `bitset`s, etc.), because this is, after all, a book on the STL.

- *Do you need to be able to insert a new element at an arbitrary position in the container?* If so, you need a sequence container; associative containers won't do.
- *Do you care how elements are ordered in the container?* If not, a hashed container becomes a viable choice. Otherwise, you'll want to avoid hashed containers.
- *Must the container be part of standard C++?* If so, that eliminates hashed containers, `slist`, and `rope`.
- *What category of iterators do you require?* If they must be random access iterators, you're technically limited to `vector`, `deque`, and `string`, but you'd probably want to consider `rope`, too. (See Item 50 for

information on `rope`.) If bidirectional iterators are required, you must avoid `slist` (see Item 50) as well as one common implementation of the hashed containers (see Item 25).

- *Is it important to avoid movement of existing container elements when insertions or erasures take place?* If so, you'll need to stay away from contiguous-memory containers (see Item 5).
- *Does the data in the container need to be layout-compatible with C?* If so, you're limited to `vector`s (see Item 16).
- *Is lookup speed a critical consideration?* If so, you'll want to look at hashed containers (see Item 25), sorted `vector`s (see Item 23), and the standard associative containers—probably in that order.
- *Do you mind if the underlying container uses reference counting?* If so, you'll want to steer clear of `string`, because many `string` implementations are reference-counted (see Item 13). You'll need to avoid `rope`, too, because the definitive `rope` implementation is based on reference counting (see Item 50). You have to represent your strings somehow, of course, so you'll want to consider `vector<char>`.
- *Do you need transactional semantics for insertions and erasures?* That is, do you require the ability to reliably roll back insertions and erasures? If so, you'll want to use a node-based container. If you need transactional semantics for multiple-element insertions (e.g., the range form—see Item 5), you'll want to choose `list`, because `list` is the only standard container that offers transactional semantics for multiple-element insertions. Transactional semantics are particularly important for programmers interested in writing exception-safe code. (Transactional semantics can be achieved with contiguous-memory containers, too, but there is a performance cost, and the code is not as straightforward. To learn more about this, consult Item 17 of Sutter's *Exceptional C++* [8].)
- *Do you need to minimize iterator, pointer, and reference invalidation?* If so, you'll want to use node-based containers, because insertions and erasures on such containers never invalidate iterators, pointers, or references (unless they point to an element you are erasing). In general, insertions or erasures on contiguous-memory containers may invalidate all iterators, pointers, and references into the container.

- *Do you care if using `swap` on containers invalidates iterators, pointers, or references?* If so, you'll need to avoid `string`, because `string` is alone in the STL in invalidating iterators, pointers, and references during `swap`s.
- *Would it be helpful to have a sequence container with random access iterators where pointers and references to the data are not invalidated as long as nothing is erased and insertions take place only at the ends of the container?* This is a very special case, but if it's *your* case, `deque` is the container of your dreams. (Interestingly, `deque`'s iterators *may* be invalidated when insertions are made only at the ends of the container. `deque` is the only standard STL container whose iterators may be invalidated without also invalidating its pointers and references.)

These questions are hardly the end of the matter. For example, they don't take into account the varying memory allocation strategies employed by the different container types. (Items 10 and 14 discuss some aspects of such strategies.) Still, they should be enough to convince you that, unless you have no interest in element ordering, standards conformance, iterator capabilities, layout compatibility with C, lookup speed, behavioral anomalies due to reference counting, the ease of implementing transactional semantics, or the conditions under which iterators are invalidated, you have more to think about than simply the algorithmic complexity of container operations. Such complexity is important, of course, but it's far from the entire story.

The STL gives you lots of options when it comes to containers. If you look beyond the bounds of the STL, there are even more options. Before choosing a container, be sure to consider *all* your options. A "default container"? I don't think so.

Item 2: Beware the illusion of container-independent code

The STL is based on generalization. Arrays are generalized into containers and parameterized on the types of objects they contain. Functions are generalized into algorithms and parameterized on the types of iterators they use. Pointers are generalized into iterators and parameterized on the type of objects they point to.

That's just the beginning. Individual container types are generalized into sequence and associative containers, and similar containers are given similar functionality. Standard contiguous-memory containers (see Item 1) offer random-access iterators, while standard node-based containers (again, see Item 1) provide bidirectional iterators. Sequence containers support `push_front` and/or `push_back`, while associative containers don't. Associative containers offer logarithmic-time `lower_bound`, `upper_bound`, and `equal_range` member functions, but sequence containers don't.

With all this generalization going on, it's natural to want to join the movement. This sentiment is laudable, and when you write your own containers, iterators, and algorithms, you'll certainly want to pursue it. Alas, many programmers try to pursue it in a different manner. Instead of committing to particular types of containers in their software, they try to generalize the notion of a container so that they can use, say, a `vector`, but still preserve the option of replacing it with something like a `deque` or a `list` later—all without changing the code that uses it. That is, they strive to write *container-independent code*. This kind of generalization, well-intentioned though it is, is almost always misguided.

Even the most ardent advocate of container-independent code soon realizes that it makes little sense to try to write software that will work with both sequence and associative containers. Many member functions exist for only one category of container, e.g., only sequence containers support `push_front` or `push_back`, and only associative containers support `count` and `lower_bound`, etc. Even such basics as `insert` and `erase` have signatures and semantics that vary from category to category. For example, when you `insert` an object into a sequence container, it stays where you put it, but if you `insert` an object into an associative container, the container moves the object to where it belongs in the container's sort order. For another example, the form of `erase` taking an iterator returns a new iterator when invoked on a sequence container, but it returns nothing when invoked on an associative container. (Item 9 gives an example of how this can affect the code you write.)

Suppose, then, you aspire to write code that can be used with the most common sequence containers: `vector`, `deque`, and `list`. Clearly, you must program to the intersection of their capabilities, and that means no uses of `reserve` or `capacity` (see Item 14), because `deque` and `list` don't offer them. The presence of `list` also means you give up `operator[]`, and you limit yourself to the capabilities of bidirectional iterators. That, in turn, means you must stay away from algorithms that demand random access iterators, including `sort`, `stable_sort`, `partial_sort`, and `nth_element` (see Item 31).

On the other hand, your desire to support `vector` rules out use of `push_front` and `pop_front`, and both `vector` and `deque` put the kibosh on `splice` and the member form of `sort`. In conjunction with the constraints above, this latter prohibition means that there is no form of `sort` you can call on your "generalized sequence container."

That's the obvious stuff. If you violate any of those restrictions, your code will fail to compile with at least one of the containers you want to be able to use. The code that *will* compile is more insidious.

The main culprit is the different rules for invalidation of iterators, pointers, and references that apply to different sequence containers. To write code that will work correctly with `vector`, `deque`, and `list`, you must assume that any operation invalidating iterators, pointers, or references in any of those containers invalidates them in the container you're using. Thus, you must assume that every call to `insert` invalidates everything, because `deque::insert` invalidates all iterators and, lacking the ability to call `capacity`, `vector::insert` must be assumed to invalidate all pointers and references. (Item 1 explains that `deque` is unique in sometimes invalidating its iterators without invalidating its pointers and references.) Similar reasoning leads to the conclusion that, unless you're `erase`ing the last element of a container, calls to `erase` must also be assumed to invalidate everything.

Want more? You can't pass the data in the container to a C interface, because only `vector` supports that (see Item 16). You can't instantiate your container with `bool` as the type of objects to be stored, because, as Item 18 explains, `vector<bool>` doesn't always behave like a `vector`, and it never actually stores `bool`s. You can't assume `list`'s constant-time insertions and erasures, because `vector` and `deque` take linear time to perform those operations.

When all is said and done, you're left with a "generalized sequence container" where you can't call `reserve`, `capacity`, `operator[]`, `push_front`, `pop_front`, `splice`, or any algorithm requiring random access iterators; a container where every call to `insert` and `erase` takes linear time and invalidates all iterators, pointers, and references; and a container incompatible with C where `bool`s can't be stored. Is that really the kind of container you want to use in your applications? I suspect not.

If you rein in your ambition and decide you're willing to drop support for `list`, you still give up `reserve`, `capacity`, `push_front`, and `pop_front`; you still must assume that all calls to `insert` and `erase` take linear time and invalidate everything; you still lose layout compatibility with C; and you still can't store `bool`s.

If you abandon the sequence containers and shoot instead for code that can work with different associative containers, the situation isn't much better. Writing for both `set` and `map` is close to impossible, because `set`s store single objects while `map`s store `pair`s of objects. Even writing for both `set` and `multiset` (or `map` and `multimap`) is tough. The `insert` member function taking only a value has different return types for `set`s/`map`s than for their `multi` cousins, and you must religiously avoid making any assumptions about how many copies of a value are stored in a container. With `map` and `multimap`, you must avoid using `operator[]`, because that member function exists only for `map`.

Face the truth: it's not worth it. The different containers are *different*, and they have strengths and weaknesses that vary in significant ways. They're not designed to be interchangeable, and there's little you can do to paper that over. If you try, you're merely tempting fate, and fate doesn't like to be tempted.

Still, the day will dawn when you'll realize that a container choice you made was, er, suboptimal, and you'll need to use a different container type. You now know that when you change container types, you'll not only need to fix whatever problems your compilers diagnose, you'll also need to examine all the code using the container to see what needs to be changed in light of the new container's performance characteristics and rules for invalidation of iterators, pointers, and references. If you switch from a `vector` to something else, you'll also have to make sure you're no longer relying on `vector`'s C-compatible memory layout, and if you switch to a `vector`, you'll have to ensure that you're not using it to store `bool`s.

Given the inevitability of having to change container types from time to time, you can facilitate such changes in the usual manner: by encapsulating, encapsulating, encapsulating. One of the easiest ways to do this is through the liberal use of typedefs for container types. Hence, instead of writing this,

Click here to view code image

```
class Widget { ... };
```

```
vector<Widget> vw;

Widget bestWidget;

...                                              // give
bestWidget a value

vector<Widget>::iterator i =                     // find a
Widget with the
  find(vw.begin(), vw.end(), bestWidget);        // same value
as bestWidget
```

write this:

[Click here to view code image]

```
class Widget { ... };

typedef vector<Widget> WidgetContainer;

WidgetContainer cw;

Widget bestWidget;

...

WidgetContainer::iterator i = find(cw.begin(), cw.end(),
bestWidget);
```

This makes it a lot easier to change container types, something that’s especially convenient if the change in question is simply to add a custom allocator. (Such a change doesn’t affect the rules for iterator/pointer/reference invalidation.)

[Click here to view code image]

```
class Widget { ... };

template<typename T>                          // see Item 10
for why this
SpecialAllocator { ... };                     // needs to be a
template

typedef vector<Widget, SpecialAllocator<Widget> >
WidgetContainer;
```

```
WidgetContainer cw;                          // still works

Widget bestWidget;

...

WidgetContainer::iterator i =
find(cw.begin(), cw.end(), bestWidget);    // still works
```

If the encapsulating aspects of typedefs mean nothing to you, you're still likely to appreciate the work they can save, especially for iterator types. For example, if you have an object of type

Click here to view code image

```
map<string,
    vector<Widget>::iterator,
    CIStringCompare>                    // CIStringCompare is
"case-
                                        // insensitive string
compare;"
                                        // Item 19 describes it
```

and you want to walk through the map using `const_iterator`s, do you really want to spell out

```
map<string, vector<Widget>::iterator,
CIStringCompare>::const_iterator
```

more than once? Once you've used the STL a little while, you'll realize that typedefs are your friends.

A typedef is just a synonym for some other type, so the encapsulation it affords is purely lexical. A typedef doesn't prevent a client from doing (or depending on) anything they couldn't already do (or depend on). You need bigger ammunition if you want to limit client exposure to the container choices you've made. You need classes.

To limit the code that may require modification if you replace one container type with another, hide the container in a class, and limit the amount of container-specific information visible through the class interface. For example, if you need to create a customer list, don't use a `list` directly. Instead, create a `CustomerList` class, and hide a `list` in its private section:

Click here to view code image

```
class CustomerList {
```

```
private:
  typedef list<Customer> CustomerContainer;
  typedef CustomerContainer::iterator CCIterator;

  CustomerContainer customers;

public:                               // limit the amount of
list-specific
  ...                                 // information visible
through
};                                    // this interface
```

At first, this may seem silly. After all a customer list is a *list*, right? Well, maybe. Later you may discover that you don't need to insert or erase customers from the middle of the list as often as you'd anticipated, but you do need to quickly identify the top 20% of your customers—a task tailor-made for the `nth_element` algorithm (see Item 31). But `nth_element` requires random access iterators. It won't work with a `list`. In that case, your customer "list" might be better implemented as a `vector` or a `deque`.

When you consider this kind of change, you still have to check every `CustomerList` member function and every `friend` to see how they'll be affected (in terms of performance and iterator/pointer/reference invalidation, etc.), but if you've done a good job of encapsulating `CustomerList`'s implementation details, the impact on `CustomerList` clients should be small. *You* can't write container-independent code, but *they* might be able to.

Item 3: Make copying cheap and correct for objects in containers

Containers hold objects, but not the ones you give them. Instead, when you add an object to a container (via, e.g., `insert` or `push_back`, etc.), what goes into the container is a *copy* of the object you specify.

Once an object is in a container, it's not uncommon for it to be copied further. If you `insert` something into or `erase` something from a `vector`, `string`, or `deque`, existing container elements are typically moved (copied) around (see Items 5 and 14). If you use any of the sorting algorithms (see Item 31); `next_permutation` or `previous_permutation`; `remove`, `unique`, or their ilk (see Item 32); `rotate` or `reverse`, etc., objects will be moved (copied) around. Yes, copying objects is the STL way.

It may interest you to know how all this copying is accomplished. That's easy. An object is copied by using its copying member functions, in particular, its *copy* constructor and its *copy* assignment operator. (Clever names, no?) For a user-defined class like `Widget`, these functions are traditionally declared like this:

Click here to view code image

```
class Widget {
public:
  ...
  Widget(const Widget&);                     // copy constructor
  Widget& operator=(const Widget&);          // copy assignment
operator
  ...
};
```

As always, if you don't declare these functions yourself, your compilers will declare them for you. Also as always, the copying of built-in types (e.g., `int`s, pointers, etc.) is accomplished by simply copying the underlying bits. (For details on copy constructors and assignment operators, consult any introductory book on C++. In *Effective C++*, Items 11 and 27 focus on the behavior of these functions.)

With all this copying taking place, the motivation for this Item should now be clear. If you fill a container with objects where copying is expensive, the simple act of putting the objects into the container could prove to be a performance bottleneck. The more things get moved around in the container, the more memory and cycles you'll blow on making copies. Furthermore, if you have objects where "copying" has an unconventional meaning, putting such objects into a container will invariably lead to grief. (For an example of the kind of grief it can lead to, see Item 8.)

In the presence of inheritance, of course, copying leads to slicing. That is, if you create a container of base class objects and you try to insert derived class objects into it, the derivedness of the objects will be removed as the objects are copied (via the base class copy constructor) into the container:

Click here to view code image

```
vector<Widget> vw;

class SpecialWidget:                          // SpecialWidget
inherits from
```

```
  public Widget { ... };                      // Widget above

SpecialWidget sw;

vw.push_back(sw);                             // sw is copied as a
base class
                                              // object into vw.
Its specialness
                                              // is lost during the
copying
```

The slicing problem suggests that inserting a derived class object into a container of base class objects is almost always an error. If you want the resulting object to *act* like a derived class object, e.g., invoke derived class virtual functions, etc., it *is* always an error. (For more background on the slicing problem, consult *Effective C++*, Item 22. For another example of where it arises in the STL, see Item 38.)

An easy way to make copying efficient, correct, and immune to the slicing problem is to create containers of *pointers* instead of containers of objects. That is, instead of creating a container of `Widget`, create a container of `Widget*`. Copying pointers is fast, it always does exactly what you expect (it copies the bits making up the pointer), and nothing gets sliced when a pointer is copied. Unfortunately, containers of pointers have their own STL-related headaches. You can read about them in Items 7 and 33. As you seek to avoid those headaches while still dodging efficiency, correctness, and slicing concerns, you'll probably discover that containers of *smart pointers* are an attractive option. To learn more about this option, turn to Item 7.

If all this makes it sound like the STL is copy-crazy, think again. Yes, the STL makes lots of copies, but it's generally designed to avoid copying objects *unnecessarily*. In fact, it's generally designed to avoid *creating* objects unnecessarily. Contrast this with the behavior of C's and C++'s only built-in container, the lowly array:

Click here to view code image

```
Widget w[maxNumWidgets];  // create an array of maxNumWidgets
                          // Widgets, default-constructing
each one
```

This constructs `maxNumWidgets Widget` objects, even if we normally expect to use only a few of them or we expect to immediately overwrite each default-constructed value with values we get from someplace else (e.g., a file). Using the STL instead of an array, we can use a `vector` that grows when it needs to:

Click here to view code image

```
vector<Widget> vw;              // create a vector with zero 
Widget
                                // objects that will expand as 
needed
```

We can also create an empty vector that contains enough space for `maxNumWidgets Widget`s, but where zero `Widget`s have been constructed:

Click here to view code image

```
vector<Widget> vw;

vw.reserve(maxNumWidgets);      // see Item 14 for details on 
reserve
```

Compared to arrays, STL containers are much more civilized. They create (by copying) only as many objects as you ask for, they do it only when you direct them to, and they use a default constructor only when you say they should. Yes, STL containers make copies, and yes, you need to understand that, but don't lose sight of the fact that they're still a big step up from arrays.

Item 4: Call `empty` instead of checking `size()` against zero

For any container `c`, writing

```
if (c.size() == 0) ...
```

is essentially equivalent to writing

```
if (c.empty()) ...
```

That being the case, you might wonder why one construct should be preferred to the other, especially in view of the fact that `empty` is typically implemented as an inline function that simply returns whether `size` returns 0.

You should prefer the construct using `empty`, and the reason is simple: `empty` is a constant-time operation for all standard containers, but for some `list` implementations, `size` may take linear time.

But what makes `list` so troublesome? Why can't it, too, offer a constant-time `size`? The answer has much to do with the range form of `list`'s unique splicing functions. Consider this code:

Click here to view code image

```
list<int> list1;
list<int> list2;

...

list1.splice(                                     // move all
nodes in list2
   list1.end(), list2,                            // from the
first occurrence
   find(list2.begin(), list2.end(), 5),           // of 5
through the last
   find(list2.rbegin(), list2.rend(), 10).base()  //
occurrence of 10 to the
);                                                // end of
list1. See Item 28
                                                  // for info
on the "base()" call
```

This code won’t work unless `list2` contains a 10 somewhere beyond a 5, but let’s assume that’s not a problem. Instead, let’s focus on this question: how many elements are in `list1` after the splice? Clearly, `list1` after the splice has as many elements as it did before the splice plus however many elements were spliced into it. But how many elements were spliced into it? As many as were in the range defined by `find(list2.begin(), list2.end(), 5)` and `find(list2.rbegin(), list2.rend(), 10).base()`. Okay, how many is that? Without traversing the range and counting them, there’s no way to know. And therein lies the problem.

Suppose you’re responsible for implementing `list`. `list` isn’t just any container, it’s a *standard* container, so you know your class will be widely used. You naturally want your implementation to be as efficient as possible. You figure that clients will commonly want to find out how many elements are in a `list`, so you’d like to make `size` a constant-time operation. You’d thus like to design `list` so it always knows how many elements it contains.

At the same time, you know that of all the standard containers, only `list` offers the ability to splice elements from one place to another without copying any data. You reason that many `list` clients will choose `list` specifically because it offers high-efficiency splicing. They know that splicing a range from one list to another can be accomplished in constant time, and you know that they know it, so you certainly want to meet their expectation that `splice` is a constant-time member function.

This puts you in a quandary. If `size` is to be a constant-time operation, each `list` member function must update the sizes of the lists on which it operates. That includes `splice`. But the only way for the range version of `splice` to update the sizes of the lists it modifies is for it to count the number of elements being spliced, and doing that would prevent it from achieving the constant-time performance you want for *it*. If you eliminate the requirement that the range form of `splice` update the sizes of the lists it's modifying, `splice` can be made constant-time, but then `size` becomes a linear-time operation. In general, it will have to traverse its entire data structure to see how many elements it contains. No matter how you look at it, something—`size` or the range form of `splice`—has to give. One or the other can be a constant-time operation, but not both.

Different list implementations resolve this conflict in different ways, depending on whether their authors choose to maximize the efficiency of `size` or the range form of `splice`. If you happen to be using a `list` implementation where a constant-time range form of `splice` was given higher priority than a constant-time `size`, you'll be better off calling `empty` than `size`, because `empty` is always a constant-time operation. Even if you're not using such an implementation, you might find yourself using such an implementation in the future. For example, you might port your code to a different platform where a different implementation of the STL is available, or you might just decide to switch to a different STL implementation for your current platform.

No matter what happens, you can't go wrong if you call `empty` instead of checking to see if `size() == 0`. So call `empty` whenever you need to know whether a container has zero elements.

Item 5: Prefer range member functions to their single-element counterparts

Quick! Given two vectors, `v1` and `v2`, what's the easiest way to make `v1`'s contents be the same as the second half of `v2`'s? Don't agonize over the definition of "half" when `v2` has an odd number of elements, just do something reasonable.

Time's up! If your answer was

```
v1.assign(v2.begin() + v2.size() / 2, v2.end());
```

or something quite similar, you get full credit and a gold star. If your answer involved more than one statement, but didn't use any kind of loop, you get nearly full credit, but no gold star. If your answer involved a loop, you've got some room for improvement, and if your answer involved multiple loops, well, let's just say that you really need this book.

By the way, if your response to the *answer* to the question included "Huh?", pay close attention, because you're going to learn something really useful.

This quiz is designed to do two things. First, it affords me an opportunity to remind you of the existence of the `assign` member function, a convenient beast that too many programmers overlook. It's available for all the standard sequence containers (`vector`, `string`, `deque`, and `list`). Whenever you have to completely replace the contents of a container, you should think of assignment. If you're just copying one container to another of the same type, `operator=` is the assignment function of choice, but as this example demonstrates, `assign` is available for the times when you want to give a container a completely new set of values, but `operator=` won't do what you want.

The second reason for the quiz is to demonstrate why range member functions are superior to their single-element alternatives. A *range member function* is a member function that, like STL algorithms, uses two iterator parameters to specify a range of elements over which something should be done. Without using a range member function to solve this Item's opening problem, you'd have to write an explicit loop, probably something like this:

Click here to view code image

```
vector<Widget> v1, v2;                 // assume v1 and v2
are vectors
                                       // of Widgets
...

v1.clear();
for (vector<Widget>::const_iterator ci = v2.begin() +
v2.size() / 2;
     ci != v2.end();
     ++ci)
  v1.push_back(*ci);
```

Item 43 examines in detail why you should try to avoid writing explicit loops, but you don't need to read that Item to recognize that writing this code is a lot more work than is writing the call to `assign`. As we'll see shortly, the loop also happens to impose an efficiency penalty, but we'll deal with that in a moment.

One way to avoid the loop is to follow the advice of Item 43 and employ an algorithm instead:

```
v1.clear();
copy(v2.begin() + v2.size() / 2, v2.end(),
back_inserter(v1));
```

Writing this is still more work than writing the call to `assign`. Furthermore, though no loop is present in this code, one certainly exists inside `copy` (see Item 43). As a result, the efficiency penalty remains. Again, I'll discuss that below. At this point, I want to digress briefly to observe that almost all uses of `copy` where the destination range is specified using an insert iterator (i.e., via `inserter`, `back_inserter`, or `front_inserter`) can be—*should* be—replaced with calls to range member functions. Here, for example, the call to `copy` can be replaced with a range version of `insert`:

```
v1.insert(v1.end(), v2.begin() + v2.size() / 2, v2.end());
```

This involves slightly less typing than the call to `copy`, but it also says more directly what is happening: data is being inserted into `v1`. The call to `copy` expresses that, too, but less directly. It puts the emphasis in the wrong place. The interesting aspect of what is happening is not that elements are being copied, it's that `v1` is having new data added to it. The `insert` member function makes that clear. The use of `copy` obscures it. There's nothing interesting about the fact that things are being copied, because the STL is *built* on the assumption that things will be copied. Copying is so fundamental to the STL, it's the topic of Item 3 in this book!

Too many STL programmers overuse `copy`, so the advice I just gave bears repeating: Almost all uses of `copy` where the destination range is specified using an insert iterator should be replaced with calls to range member functions.

Returning to our `assign` example, we've already identified two reasons to prefer range member functions to their single-element counterparts:

- It's generally less work to write the code using the range member functions.
- Range member functions tend to lead to code that is clearer and more

straightforward.

In short, range member functions yield code that is easier to write and easier to understand. What’s not to like?

Alas, some will dismiss these arguments as matters of programming style, and developers enjoy arguing about style issues almost as much as they enjoy arguing about which is the One True Editor. (As if there’s any doubt. It’s Emacs.) It would be helpful to have a more universally agreed-upon criterion for establishing the superiority of range member functions to their single-element counterparts. For the standard sequence containers, we have one: efficiency. When dealing with the standard sequence containers, application of single-element member functions makes more demands on memory allocators, copies objects more frequently, and/or performs redundant operations compared to range member functions that achieve the same end.

For example, suppose you’d like to copy an array of `int`s into the front of a `vector`. (The data might be in an array instead of a `vector` in the first place, because the data came from a legacy C API. For a discussion of the issues that arise when mixing STL containers and C APIs, see Item 16.) Using the `vector` range `insert` function, it’s honestly trivial:

Click here to view code image

```
int data[numValues];                             // assume
numValues is
                                                 // defined
elsewhere
vector<int> v;

...

v.insert(v.begin(), data, data + numValues);     // insert
the ints in data
                                                 // into v at
the front
```

Using iterative calls to `insert` in an explicit loop, it would probably look more or less like this:

Click here to view code image

```
vector<int>::iterator insertLoc(v.begin());
for (int i = 0; i < numValues; ++i) {
  insertLoc = v.insert(insertLoc, data[i]);
  ++insertLoc;
}
```

Notice how we have to be careful to save the return value of `insert` for the next loop iteration, then increment the returned iterator. If we didn't update `insertLoc` after each insertion, we'd have two problems. First, all loop iterations after the first would yield undefined behavior, because each `insert` call would invalidate `insertLoc`. Second, even if `insertLoc` remained valid, we'd always insert at the front of the vector (i.e., at `v.begin()`), and the result would be that the `int`s copied into `v` would end up in reverse order.

If we follow the lead of Item 43 and replace the loop with a call to `copy`, we get something like this:

```
copy(data, data + numValues, inserter(v, v.begin()));
```

By the time the `copy` template has been instantiated, the code based on `copy` and the code using the explicit loop will be almost identical, so for purposes of an efficiency analysis, we'll focus on the explicit loop, keeping in mind that the analysis is equally valid for the code employing `copy`. Looking at the explicit loop just makes it easier to understand where the efficiency hits come from. Yes, that's "hits," plural, because the code using the single-element version of `insert` levies up to three different performance taxes on you, none of which you pay if you use the range version of `insert`.

The first tax consists of unnecessary function calls. Inserting `numValues` elements into `v` one at a time naturally costs you `numValues` calls to `insert`. Using the range form of `insert`, you pay for only one function call, a savings of `numValues-1` calls. Of course, it's possible that inlining will save you from this tax, but then again, it's possible that it won't. Only one thing is sure. With the range form of `insert`, you definitely won't pay it.

Inlining won't save you from the second tax, which is the cost of inefficiently moving the existing elements in `v` to their final post-insertion positions. Each time `insert` is called to add a new value to `v`, every element above the insertion point must be moved up one position to make room for the new element. So the element at position *p* must be moved up to position *p+1*, etc. In our example, we're inserting `numValues` elements at the front of `v`. That means that each element in `v` prior to the insertions will have to be shifted up a total of `numValues` positions. But each will be shifted up only one position each time `insert` is called, so each element will be moved a total of `numValues` times. If `v` has *n* elements prior to the insertions, a total of *n**`numValues` moves will take place. In this example, `v` holds `int`s, so each move will probably boil down to an invocation of `memmove`, but if `v` held a user-defined type like `Widget`, each move would incur a call to that type's assignment operator or copy constructor. (Most calls would be to the assignment operator, but each time the last element in the vector was moved, that move would be accomplished by calling the element's copy constructor.) In the general case, then, inserting `numValues` new objects one at a time into the front of a `vector<Widget>` holding *n* elements exacts a cost of *n**`numValues` function calls: (*n*-1)*`numValues` calls to the `Widget` assignment operator and `numValues` calls to the `Widget` copy constructor. Even if these calls are inlined, you're still doing the work to move the elements in `v` `numValues` times.

In contrast, the Standard requires that range `insert` functions move existing container elements directly into their final positions, i.e., at a cost of one move per element. The total cost is *n* moves, `numValues` to the copy constructor for the type of objects in the container, the remainder to that type's assignment operator. Compared to the single-element `insert` strategy, the range `insert` performs *n**(`numValues`-1) fewer moves. Think about that for a minute. It means that if `numValues` is 100, the range form of `insert` would do 99% fewer moves than the code making repeated calls to the single-element form of `insert`!

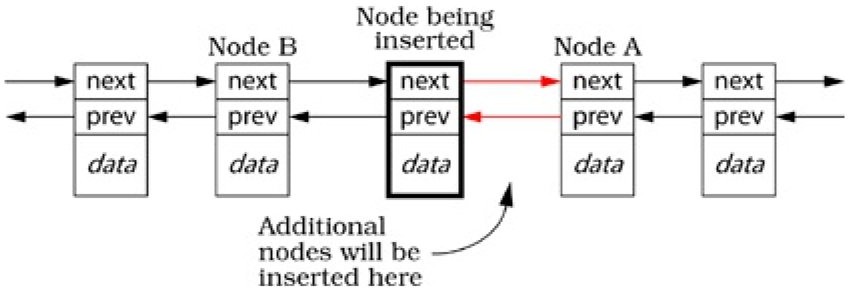
Before I move on to the third efficiency cost of single-element member functions vis-à-vis their range counterparts, I have a minor correction. What I wrote in the previous paragraph is the truth and nothing but the truth, but it's not quite the whole truth. A range `insert` function can move an element into its final position in a single move only if it can determine the distance between two iterators without losing its place. This is almost always possible, because all forward iterators offer this functionality, and forward iterators are nearly ubiquitous. All iterators for the standard containers offer forward iterator functionality. So do the iterators for the nonstandard hashed containers (see Item 25). Pointers acting as iterators into arrays offer such functionality, too. In fact, the only standard iterators that don't offer forward iterator capabilities are input and output iterators. Thus, everything I wrote above is true except when the iterators passed to the range form of `insert` are input iterators (e.g. `istream_iterator`s—see Item 6). In that case only, range `insert` is allowed to move elements into their final positions one place at a time, and if it's implemented that way, its advantage as regards number of element movements ceases to exist. (For output iterators, this issue fails to arise, because output iterators can't be used to specify a range for `insert`.)

The final performance tax levied on those so foolish as to use repeated single-element insertions instead of a single range insertion has to do with memory allocation, though it has a nasty copying side to it, too. As Item 14 explains, when you try to insert an element into a vector whose memory is full, the vector allocates new memory with more capacity, copies its elements from the old memory to the new memory, destroys the elements in the old memory, and deallocates the old memory. Then it adds the element that is being inserted. Item 14 also explains that vector implementations increase their capacity by some multiplicative factor each time they run out of memory, so inserting `numValues` new elements often results in new memory being allocated a number of times proportional to the logarithm of `numValues`. Inserting 1000 elements one at a time into a vector with a growth rate of 1.5 could thus result in 18 new allocations[†] (including their incumbent copying of elements). In contrast (and, by now, predictably), a range insertion can figure out how much new memory it needs before it starts inserting things (assuming it is given forward iterators), so it need not reallocate a `vector`'s underlying memory more than once. As you can imagine, the savings can be considerable.

The analysis I've just performed is for `vector`s, but the same reasoning applies to `string`s, too. For `deque`s, the reasoning is similar, but `deque`s manage their memory differently from `vector`s and `string`s, so the argument about repeated memory allocations doesn't apply. The argument about moving elements an unnecessarily large number of times, however, generally does apply (though the details are different), as does the observation about the number of function calls.

Among the standard sequence containers, that leaves only `list`, but here, too, there is a performance advantage to using a range form of `insert` instead of a single-element form. The argument about repeated function calls continues to be valid, of course, but, because of the way linked lists work, the copying and memory allocation issues fail to arise. Instead, there is a new problem: repeated superfluous assignments to the `next` and `prev` pointers of some nodes in the list.

Each time an element is added to a linked list, the list node holding that element must have its `next` and `prev` pointers set, and of course the node preceding the new node (let’s call it B, for “before”) must set its `next` pointer and the node following the new node (we’ll call it A, for “after”) must set its `prev` pointer:



When a series of new nodes is added one by one by calling `list`’s single-element `insert`, all but the last new node will set its `next` pointer *twice*, once to point to A, a second time to point to the element inserted after it. A will set its `prev` pointer to point to a new node each time one is inserted in front of it. If `numValues` nodes are inserted in front of A, `numValues`-1 superfluous assignments will be made to the inserted nodes’ `next` pointers, and `numValues`-1 superfluous assignments will be made to A’s `prev` pointer. All told, that’s 2*(`numValues`-1) unnecessary pointer assignments. Pointer assignments are cheap, of course, but why pay for them if you don’t have to?

By now it should be clear that you don’t have to, and the key to evading the cost is to use `list`’s range form of `insert`. Because that function knows how many nodes will ultimately be inserted, it can avoid the superfluous pointer assignments, using only a single assignment to each pointer to set it to its proper post-insertion value.

For the standard sequence containers, then, a lot more than programming style is on the line when choosing between single-element insertions and range insertions. For the associative containers, the efficiency case is harder to make, though the issue of extra function call overhead for repeated calls to single-element `insert` continues to apply. Furthermore, certain special kinds of range insertions may lead to optimization possibilities in associative containers, too, but as far as I can tell, such optimizations currently exist only in theory. By the time you read this, of course, theory may have become practice, so range insertions into associative containers may indeed be more efficient than repeated single-element insertions. Certainly they are never *less* efficient, so you have nothing to lose by preferring them.

Even without the efficiency argument, the fact remains that using range member functions requires less typing as you write the code, and it also yields code that is easier to understand, thus enhancing your software's long-term maintainability. Those two characteristics alone should convince you to prefer range member functions. The efficiency edge is really just a bonus.

Having droned on this long about the wonder of range member functions, it seems only appropriate that I summarize them for you. Knowing which member functions support ranges makes it a lot easier to recognize opportunities to use them. In the signatures below, the parameter type `iterator` literally means the iterator type for the container, i.e, *container*::`iterator`. The parameter type `InputIterator`, on the other hand, means that any input iterator is acceptable.

- **Range construction.** All standard containers offer a constructor of this form:

```
container::container(InputIterator begin,   // beginning
of range
                     InputIterator end);    // end of
range
```

  When the iterators passed to this constructor are `istream_iterator`s or `istreambuf_iterator`s (see Item 29), you may encounter C++'s most astonishing parse, one that causes your compilers to interpret this construct as a function declaration instead of as the definition of a new container object. Item 6 tells you everything you need to know about that parse, including how to defeat it.

- **Range insertion.** All standard sequence containers offer this form of `insert`:

Click here to view code image

```
void container::insert(iterator position,     // where
to insert the range
                       InputIterator begin,   // start
of range to insert
                       InputIterator end);    // end of
range to insert
```

  Associative containers use their comparison function to determine where elements go, so they offer a signature that omits the position parameter:

```
void container::insert(InputIterator begin,
InputIterator end);
```

When looking for ways to replace single-element `insert`s with range versions, don’t forget that some single-element variants camouflage themselves by adopting different function names. For example, `push_front` and `push_back` both insert single elements into containers, even though they’re not called `insert`. If you see a loop calling `push_front` or `push_back`, or if you see an algorithm such as `copy` being passed `front_inserter` or `back_inserter` as a parameter, you’ve discovered a place where a range form of `insert` is likely to be a superior strategy.

- **Range erasure.** Every standard container offers a range form of `erase`, but the return types differ for sequence and associative containers. Sequence containers provide this,

```
iterator container::erase(iterator begin, iterator end);
```

while associative containers offer this:

```
void container::erase(iterator begin, iterator end);
```

Why the difference? The claim is that having the associative container version of `erase` return an iterator (to the element following the one that was erased) would incur an unacceptable performance penalty. I’m one of many who find this claim specious, but the Standard says what the Standard says, and what the Standard says is that sequence and associative container versions of `erase` have different return types.

Most of this Item’s efficiency analysis for `insert` has analogues for `erase`. The number of function calls is still greater for repeated calls to single-element `erase` than for a single call to range `erase`. Element values must still be shifted one position at a time towards their final destination when using single-element `erase`, while range `erase` can move them into their final positions in a single move.

One argument about `vector`’s and `string`’s `insert` that fails to apply to `erase` has to do with repeated allocations. (For `erase`, of course, it would concern repeated *de*allocations.) That’s because the memory for `vector`s and `string`s automatically grows to accommodate new elements, but it doesn’t automatically shrink when the number of elements is reduced. (Item 17 describes how you may reduce the unnecessary memory held by a `vector` or `string`.)

One particularly important manifestation of range `erase` is the `erase-remove` idiom. You can read all about it in Item 32.

- **Range assignment.** As I noted at the beginning of this Item, all standard sequence containers offer a range form of `assign`:

  ```
  void container::assign(InputIterator begin,
  InputIterator end);
  ```

So there you have it, three solid arguments for preferring range member functions to their single-element counterparts. Range member functions are easier to write, they express your intent more clearly, and they exhibit higher performance. That's a troika that's hard to beat.

Item 6: Be alert for C++'s most vexing parse

Suppose you have a file of `int`s and you'd like to copy those ints into a `list`. This seems like a reasonable way to do it:

Click here to view code image

```
ifstream dataFile("ints.dat");

list<int> data(istream_iterator<int>(dataFile),  // warning!
this doesn't do
               istream_iterator<int>());         // what you
think it does
```

The idea here is to pass a pair of `istream_iterator`s to `list`'s range constructor (see Item 5), thus copying the ints in the file into the list.

This code will compile, but at runtime, it won't do anything. It won't read any data out of a file. It won't even create a `list`. That's because the second statement doesn't declare a `list` and it doesn't call a constructor. What it does is ... well, what it does is so strange, I dare not tell you straight out, because you won't believe me. Instead, I have to develop the explanation, bit by bit. Are you sitting down? If not, you might want to look around for a chair...

We'll start with the basics. This line declares a function `f` taking a `double` and returning an `int`:

```
int f(double d);
```

This next line does the same thing. The parentheses around the parameter name `d` are superfluous and are ignored:

```
int f(double (d));         // same as above; parens around d
are ignored
```

The line below declares the same function. It simply omits the parameter name:

```
int f(double);            // same as above; parameter name is
omitted
```

Those three declaration forms should be familiar to you, though the ability to put parentheses around a parameter name may have been new. (It wasn’t long ago that it was new to me.)

Let’s now look at three more function declarations. The first one declares a function `g` taking a parameter that’s a pointer to a function taking nothing and returning a `double`:

```
int g(double (*pf)());    // g takes a pointer to a function
as a parameter
```

Here’s another way to say the same thing. The only difference is that `pf` is declared using non-pointer syntax (a syntax that’s valid in both C and C++):

```
int g(double pf());       // same as above; pf is implicitly
a pointer
```

As usual, parameter names may be omitted, so here’s a third declaration for `g`, one where the name `pf` has been eliminated:

```
int g(double ());         // same as above; parameter name is
omitted
```

Notice the difference between parentheses *around a parameter name* (such as `d` in the second declaration for `f`) and *standing by themselves* (as in this example). Parentheses around a parameter name are ignored, but parentheses standing by themselves indicate the existence of a parameter list; they announce the presence of a parameter that is itself a pointer to a function.

Having warmed ourselves up with these declarations for `f` and `g`, we are ready to examine the code that began this Item. Here it is again:

```
list<int> data(istream_iterator<int>(dataFile),
               istream_iterator<int>());
```

Brace yourself. This declares a *function*, `data`, whose return type is `list<int>`. The function `data` takes two parameters:

- The first parameter is named `dataFile`. Its type is `istream_iterator<int>`. The parentheses around `dataFile` are superfluous and are ignored.
- The second parameter has no name. Its type is pointer to function taking nothing and returning an `istream_iterator<int>`.

Amazing, huh? But it’s consistent with a universal rule in C++, which says that pretty much anything that can be parsed as a function declaration will be. If you’ve been programming in C++ for a while, you’ve almost certainly encountered another manifestation of this rule. How many times have you seen this mistake?

Click here to view code image

```
class Widget { ... };            // assume Widget has a
default constructor

Widget w();                      // uh oh...
```

This doesn’t declare a `Widget` named `w`, it declares a function named `w` that takes nothing and returns a `Widget`. Learning to recognize this *faux pas* is a veritable rite of passage for C++ programmers.

All of which is interesting (in its own twisted way), but it doesn’t help us say what we want to say, which is that a `list<int>` object should be initialized with the contents of a file. Now that we know what parse we have to defeat, that’s easy to express. It’s not legal to surround a formal parameter declaration with parentheses, but it is legal to surround an argument to a function call with parentheses, so by adding a pair of parentheses, we force compilers to see things our way:

Click here to view code image

```
list<int> data((istream_iterator<int>(dataFile)),  // note
new parens
               istream_iterator<int>());           // around
first argument
                                                   // to
list's constructor
```

This is the proper way to declare `data`, and given the utility of `istream_iterator`s and range constructors (again, see Item 5), it’s worth knowing how to do it.

Unfortunately, not all compilers currently know it themselves. Of the several I tested, almost half refused to accept `data`’s declaration unless it was *incorrectly* declared without the additional parentheses! To placate such compilers, you could roll your eyes and use the declaration for `data` that I’ve painstakingly explained is incorrect, but that would be both unportable and short-sighted. After all, compilers that currently get the parse wrong will surely correct it in the future, right? (Surely!)

A better solution is to step back from the trendy use of anonymous `istream_iterator` objects in `data`'s declaration and simply give those iterators names. The following code should work everywhere:

Click here to view code image

```
ifstream dataFile("ints.dat");

istream_iterator<int> dataBegin(dataFile);
istream_iterator<int> dataEnd;

list<int> data(dataBegin, dataEnd);
```

This use of named iterator objects runs contrary to common STL programming style, but you may decide that's a price worth paying for code that's unambiguous to both compilers and the humans who have to work with them.

Item 7: When using containers of `new`ed pointers, remember to `delete` the pointers before the container is destroyed

Containers in the STL are remarkably smart. They serve up iterators for both forward and reverse traversals (via `begin`, `end`, `rbegin`, etc.); they tell you what type of objects they contain (via their `value_type` typedef); during insertions and erasures, they take care of any necessary memory management; they report both how many objects they hold and the most they may contain (via `size` and `max_size`, respectively); and of course they automatically destroy each object they hold when they (the containers) are themselves destroyed.

Given such brainy containers, many programmers stop worrying about cleaning up after themselves. Heck, they figure, their containers will do the worrying for them. In many cases, they're right, but when the containers hold *pointers* to objects allocated with `new`, they're not right enough. Sure, a container of pointers will destroy each element it contains when it (the container) is destroyed, but the "destructor" for a pointer is a no-op! It certainly doesn't call `delete`.

As a result, the following code leads straight to a resource leak:

Click here to view code image

```
void doSomething()
{
  vector<Widget*> vwp;
  for (int i = 0; i < SOME_MAGIC_NUMBER; ++i)
    vwp.push_back(new Widget);
  ...                                     // use vwp
}                                         // Widgets are leaked
here!
```

Each of `vwp`'s elements is destroyed when `vwp` goes out of scope, but that doesn't change the fact that `delete` was never used for the objects conjured up with `new`. Such deletion is your responsibility, not that of your vector. This is a feature. Only you know whether the pointers *should* be deleted.

Usually, you want them to be. When that's the case, making it happen seems easy enough:

Click here to view code image

```
void doSomething()
{
  vector<Widget*> vwp;

  ...                                      // as before

  for(vector<Widget*>::iterator i = vwp.begin();
      i != vwp.end();
      ++i)
    delete *i;
}
```

This works, but only if you're not terribly picky about what you mean by "works". One problem is that the new `for` loop does pretty much what `for_each` does, but it's not as clear as using `for_each` (see Item 43). Another is that the code isn't exception safe. If an exception is thrown between the time `vwp` is filled with pointers and the time you get around to `delete`ing them, you've leaked resources again. Fortunately, both problems can be overcome.

To turn your `for_each`-like loop into an actual use of `for_each`, you need to turn `delete` into a function object. That's child's play, assuming you have a child who likes to play with the STL:

Click here to view code image

```
template<typename T>
struct DeleteObject:                                  // Item 40
```

```
describes why
  public unary_function<const T*, void> {          // this
inheritance is here

  void operator()(const T* ptr) const
  {
    delete ptr;
  }
};
```

Now you can do this:

Click here to view code image

```
void doSomething()
{
  ...                                         // as before

  for_each(vwp.begin(), vwp.end(), DeleteObject<Widget>());
}
```

Unfortunately, this makes you specify the type of objects that `DeleteObject` will be deleting (in this case, `Widget`). That's annoying. `vwp` is a `vector<Widget*>`, so *of course* `DeleteObject` will be deleting `Widget*` pointers! Duh! This kind of redundancy is more than just annoying, because it can lead to bugs that are difficult to track down. Suppose, for example, somebody ill-advisedly decides to inherit from `string`:

```
class SpecialString: public string { ... };
```

This is risky from the get-go, because `string`, like all the standard STL containers, lacks a virtual destructor, and publicly inheriting from classes without virtual destructors is a major C++ no-no. (For details, consult any good book on C++. In *Effective C++*, the place to look is Item 14.) Still, some people do this kind of thing, so let's consider how the following code would behave:

Click here to view code image

```
void doSomething()
{
  deque<SpecialString*> dssp;

  ...

  for_each(dssp.begin(), dssp.end(),    // undefined behavior!
Deletion
```

```
           DeleteObject<string>());    // of a derived object
via a base
}                                      // class pointer where
there is
                                       // no virtual
destructor
```

Note how `dssp` is declared to hold `SpecialString*` pointers, but the author of the `for_each` loop has told `DeleteObject` that it will be deleting `string*` pointers. It’s easy to understand how such an error could arise. `SpecialString` undoubtedly acts a lot like a `string`, so one can forgive its clients if they occasionally forget that they are using `SpecialString`s instead of `string`s.

We can eliminate the error (as well as reduce the number of keystrokes required of `DeleteObject`’s clients) by having compilers deduce the type of pointer being passed to `DeleteObject::operator()`. All we need to do is move the templatization from `DeleteObject` to its `operator()`:

Click here to view code image

```
struct DeleteObject {                     // templatization
and base
                                          // class removed
here

  template<typename T>                    // templatization
added here
  void operator()(const T* ptr) const
  {
    delete ptr;
  }
};
```

Compilers know the type of pointer being passed to `DeleteObject::operator()`, so we have them automatically instantiate an `operator()` taking that type of pointer. The downside to this type deduction is that we give up the ability to make `DeleteObject` adaptable (see Item 40). Considering how `DeleteObject` is designed to be used, it’s difficult to imagine how that could be a problem.

With this new version of `DeleteObject`, the code for `SpecialString` clients looks like this:

Click here to view code image

```
void doSomething()
{
  deque<SpecialString*> dssp;
  ...

  for_each(dssp.begin(), dssp.end(),
           DeleteObject());                    // ah! well-defined
behavior!
}
```

Straightforward and type-safe, just the way we like it.

But still not exception-safe. If an exception is thrown after the `SpecialString`s are `new`ed but before invocation of the call to `for_each`, it's Leakapalooza. That problem can be addressed in a variety of ways, but the simplest is probably to replace the container of pointers with a container of *smart pointers*, typically reference-counted pointers. (If you're unfamiliar with the notion of smart pointers, you should be able to find a description in any intermediate or advanced C++ book. In *More Effective C++*, the material is in Item 28.)

The STL itself contains no reference-counting smart pointer, and writing a good one—one that works correctly all the time—is tricky enough that you don't want to do it unless you have to. I published the code for a reference-counting smart pointer in *More Effective C++* in 1996, and despite basing it on established smart pointer implementations and submitting it to extensive pre-publication reviewing by experienced developers, a small parade of valid bug reports has trickled in for years. The number of subtle ways in which reference-counting smart pointers can fail is remarkable. (For details, consult the *More Effective C++* errata list [28].)

Fortunately, there's rarely a need to write your own, because proven implementations are not difficult to find. One such smart pointer is `shared_ptr` in the Boost library (see Item 50). With Boost's `shared_ptr`, this Item's original example can be rewritten as follows:

Click here to view code image

```
void doSomething()
{
  typedef boost::shared_ptr<Widget> SPW;       // SPW =
"shared_ptr
                                               //        to
Widget"
  vector<SPW> vwp;
  for (int i = 0; i < SOME_MAGIC_NUMBER; ++i)
    vwp.push_back(SPW(new Widget));            // create an
SPW from a
                                              // Widget*,
then do a
                                              // push_back on
it

  ...                                         // use vwp

}                                   // no Widgets are leaked
here, not
                                    // even if an exception
is thrown
                                    // in the code above
```

One thing you must *never* be fooled into thinking is that you can arrange for pointers to be deleted automatically by creating containers of `auto_ptr`s. That's a horrible thought, one so perilous, I've devoted Item 8 to why you should avoid it.

All you really need to remember is that STL containers are smart, but they're not smart enough to know whether to delete the pointers they contain. To avoid resource leaks when you have containers of pointers that should be deleted, you must either replace the pointers with smart reference-counting pointer objects (such as Boost's `shared_ptr`) or you must manually delete each pointer in the container before the container is destroyed.

Finally, it may have crossed your mind that if a struct like `DeleteObject` can make it easier to avoid resource leaks for containers holding pointers to objects, it should be possible to create a similar `DeleteArray` struct to make it easier to avoid resource leaks for containers holding pointers to arrays. Certainly it is *possible*, but whether it is advisable is a different matter. Item 13 explains why dynamically allocated arrays are almost always inferior to `vector` and `string` objects, so before you sit down to write `DeleteArray`, please review Item 13 first. With luck, you'll decide that `DeleteArray` is a struct whose time will never come.

Item 8: Never create containers of `auto_ptr`s

Frankly, this Item shouldn't need to be in *Effective STL*. Containers of `auto_ptr` (COAPs) are prohibited. Code attempting to use them shouldn't compile. The C++ Standardization Committee expended untold effort to arrange for that to be the case.[†] I shouldn't have to say anything about

COAPs, because your compilers should have plenty to say about such containers, and all of it should be uncomplimentary.

Alas, many programmers use STL platforms that fail to reject COAPs. Alas even more, many programmers see in COAPs the chimera of a simple, straightforward, efficient solution to the resource leaks that often accompany containers of pointers (see Items 7 and 33). As a result, many programmers are tempted to use COAPs, even though it's not supposed to be possible to create them.

I'll explain in a moment why the spectre of COAPs was so alarming that the Standardization Committee took specific steps to make them illegal. Right now, I want to focus on a disadvantage that requires no knowledge of `auto_ptr`, or even of containers: COAPs aren't portable. How could they be? The Standard for C++ forbids them, and better STL platforms already enforce this. It's reasonable to assume that as time goes by, STL platforms that currently fail to enforce this aspect of the Standard will become more compliant, and when that happens, code that uses COAPs will be even less portable than it is now. If you value portability (and you should), you'll reject COAPs simply because they fail the portability test.

But maybe you're not of a portability mind-set. If that's the case, kindly allow me to remind you of the unique—some would say bizarre—definition of what it means to copy an `auto_ptr`.

When you copy an `auto_ptr`, ownership of the object pointed to by the `auto_ptr` is transferred to the copying `auto_ptr`, and the copied `auto_ptr` is set to `NULL`. You read that right: *to copy an* `auto_ptr` *is to change its value*:

Click here to view code image

```
auto_ptr<Widget> pw1(new Widget);  // pw1 points to a Widget

auto_ptr<Widget> pw2(pw1);         // pw2 points to pw1's
Widget;
                                   // pw1 is set to NULL.
(Ownership
                                   // of the Widget is
transferred
                                   // from pw1 to pw2.)

pw1 = pw2;                         // pw1 now points to the
Widget
```

```
                                   // again; pw2 is set to
NULL
```

This is certainly unusual, and perhaps it's interesting, but the reason you (as a user of the STL) care is that it leads to some *very* surprising behavior. For example, consider this innocent-looking code, which creates a vector of `auto_ptr<Widget>` and then sorts it using a function that compares the values of the pointed-to `Widget`s:

Click here to view code image

```
bool widgetAPCompare(const auto_ptr<Widget>& lhs,
                     const auto_ptr<Widget>& rhs)
{
  return *lhs < *rhs;                        // for this example,
assume that
}                                            // operator< exists
for Widgets

vector<auto_ptr<Widget> > widgets;           // create a vector
and then fill it
...                                          // with auto_ptrs to
Widgets;
                                             // remember that this
should
                                             // not compile!

sort(widgets.begin(), widgets.end(),         // sort the vector
     widgetAPCompare);
```

Everything here looks reasonable, and conceptually, everything *is* reasonable, but the results need not be reasonable at all. For example, one or more of the `auto_ptr`s in `widgets` may have been set to `NULL` during the sort. The act of sorting the vector may have changed its contents! It is worthwhile understanding how this can be.

It can be because one approach to implementing `sort`—a common approach, as it turns out—is to use some variation on the quicksort algorithm. The fine points of quicksort need not concern us, but the basic idea is that to sort a container, some element of the container is chosen as the "pivot element," then a recursive sort is done on the values greater than and less than or equal to the pivot element. Within `sort`, such an approach could look something like this:

Click here to view code image

```
template<class RandomAccessIterator,       // this
declaration for
         class Compare>                    // sort is copied
straight
void sort(RandomAccessIterator first,      // out of the
Standard
          RandomAccessIterator last,
          Compare comp)
{
  // this typedef is described below
  typedef typename
iterator_traits<RandomAccessIterator>::value_type
    ElementType;

  RandomAccessIterator i;

  ...                                // make i point to the
pivot element

  ElementType pivotValue(*i);        // copy the pivot element
into a
                                     // local temporary
variable; see
                                     // discussion below

  ...                                // do the rest of the
sorting work
}
```

Unless you're an experienced reader of STL source code, this may look intimidating, but it's really not that bad. The only tricky part is the reference to `iterator_traits<RandomAccessIterator>::value_type`, and that's just the fancy STL way of referring to the type of object pointed to by the iterators passed to `sort`. (When we refer to `iterator_traits<RandomAccessIterator>::value_type`, we must precede it by `typename`, because it's the name of a type that's dependent on a template parameter, in this case, `RandomAccessIterator`. For more information about this use of `typename`, turn to page 7.)

The troublesome statement in the code above is this one,

```
ElementType pivotValue(*i);
```

because it copies an element from the range being sorted into a local temporary object. In our case, the element is an `auto_ptr<Widget>`, so this act of copying silently sets the copied `auto_ptr`—the one in the vector—to `NULL`.

Furthermore, when `pivotValue` goes out of scope, it will automatically delete the `Widget` it points to. By the time the call to `sort` returns, the contents of the vector will have changed, and at least one `Widget` will have been deleted. It's possible that several vector elements will have been set to `NULL` and several `Widget`s will have been deleted, because quicksort is a recursive algorithm, so it could well have copied a pivot element at each level of recursion.

This is a nasty trap to fall into, and that's why the Standardization Committee worked so hard to make sure you're not supposed to be able to fall into it. Honor its work on your behalf, then, by never creating containers of `auto_ptr`s, even if your STL platforms allow it.

If your goal is a container of smart pointers, this doesn't mean you're out of luck. Containers of smart pointers are fine, and Item 50 describes where you can find smart pointers that mesh well with STL containers. It's just that `auto_ptr` is not such a smart pointer. Not at all.

Item 9: Choose carefully among erasing options

Suppose you have a standard STL container, `c`, that holds `int`s,

```
Container<int> c;
```

and you'd like to get rid of all the objects in `c` with the value 1963. Surprisingly, the way to accomplish this task varies from container type to container type; no single approach works for all of them.

If you have a contiguous-memory container (`vector`, `deque`, or `string`—see Item 1), the best approach is the `erase-remove` idiom (see Item 32):

Click here to view code image

```
c.erase(remove(c.begin(), c.end(), 1963),   // the erase-
remove idiom is
        c.end());                           // the best way
to get rid of
                                            // elements with
a specific
                                            // value when c
is a vector,
                                            // string, or
deque
```

This approach works for `list`s, too, but, as Item 44 explains, the `list` member function `remove` is more efficient:

Click here to view code image

```
c.remove(1963);              // the remove member function is
the
                             // best way to get rid of
elements with
                             // a specific value when c is a
list
```

When `c` is a standard associative container (i.e., a `set`, `multiset`, `map`, or `multimap`), the use of anything named `remove` is completely wrong. Such containers have no member function named `remove`, and using the `remove` algorithm might overwrite container values (see Item 32), potentially corrupting the container. (For details on such corruption, consult Item 22, which also explains why trying to use `remove` on `map`s and `multimap`s won’t compile, and trying to use it on `set`s and `multiset`s may not compile.)

No, for associative containers, the proper way to approach the problem is to call `erase`:

Click here to view code image

```
c.erase(1963);                 // the erase member function
is the
                               // best way to get rid of
elements with
                               // a specific value when c is
a
                               // standard associative
container
```

Not only does this do the right thing, it takes only logarithmic time for `set` and `map`, and, for `multiset` and `multimap`, time linear in the number of elements with the specified value. (The `remove`-based techniques for sequence containers require time linear in the number of elements in the container.) Furthermore, the associative container `erase` member function has the advantage of being based on equivalence instead of equality, a distinction whose importance is explained in Item 19.

Let’s now revise the problem slightly. Instead of getting rid of every object in `c` that has a particular value, let’s eliminate every object for which the following predicate (see Item 39) returns `true`:

```
bool badValue(int x);            // returns whether x is
"bad"
```

For the sequence containers (`vector`, `string`, `deque`, and `list`), all we need to do is replace each use of `remove` with `remove_if`, and we're done:

Click here to view code image

```
c.erase(remove_if(c.begin(), c.end(), badValue),     // this
is the best way to
        c.end());                                    // get
rid of objects
                                                     // where
badValue
                                                     //
returns true when c is
                                                     // a
vector, string, or
                                                     // deque
c.remove_if(badValue);                  // this is the best
way to get rid of
                                        // objects where
badValue returns
                                        // true when c is a
list
```

For the standard associative containers, it's not quite so straightforward. There are two ways to approach the problem, one easier to code, one more efficient. The easier-but-less-efficient solution uses `remove_copy_if` to copy the values we want into a new container, then `swap`s the contents of the original container with those of the new one:

Click here to view code image

```
AssocContainer<int> c;                        // c is now one
of the
...                                           // standard
associative
                                              // containers

AssocContainer<int> goodValues;               // temporary
container
                                              // to hold
unremoved
                                              // values

remove_copy_if(c.begin(), c.end(),            // copy desired
               inserter(goodValues,           // values from
c to
```

```
                        goodValues.end()),    // goodValues
               badValue);

c.swap(goodValues);                           // swap the
contents of
                                              // c and
goodValues
```

The drawback to this approach is that it involves copying all the elements that aren’t being removed, and such copying might cost us more than we’re interested in paying.

We can dodge that bill by removing the elements from the original container directly. However, because associative containers offer no member function akin to `remove_if`, we must write a loop to iterate over the elements in `c`, erasing elements as we go.

Conceptually, the task is simple, and in fact, the code is simple, too. Unfortunately, the code that does the job correctly is rarely the code that springs to mind. For example, this is what many programmers come up with first:

Click here to view code image

```
AssocContainer<int> c;
...

for (AssocContainer<int>::iterator i = c.begin();    //
clear, straightforward,
     i != c.end();                                   // and
buggy code to
     ++i) {                                          // erase
every element
  if (badValue(*i)) c.erase(i);                      // in c
where badValue
}                                                    //
returns true; don't
                                                     // do
this!
```

Alas, this has undefined behavior. When an element of a container is `erase`d, all iterators that point to that element are invalidated. Once `c.erase(i)` returns, `i` has been invalidated. That’s bad news for this loop, because after `erase` returns, `i` is incremented via the `++i` part of the `for` loop.

To avoid this problem, we have to make sure we have an iterator to the next element of `c` before we call `erase`. The easiest way to do that is to use postfix increment on `i` when we make the call:

[Click here to view code image]

```
AssocContainer<int> c;
...

for (AssocContainer<int>::iterator i = c.begin();     // the
3rd part of the for
     i != c.end();                                    // loop
is empty; i is now
     /* nothing */) {                                 //
incremented below

    if (badValue(*i)) c.erase(i++);                   // for
bad values, pass the
    else ++i;                                         //
current i to erase and
}                                                     //
increment i as a side
                                                      //
effect; for good values,
                                                      // just
increment i
```

This approach to calling `erase` works, because the value of the expression `i++` is `i`’s old value, but as a side effect, `i` is incremented. Hence, we pass `i`’s old (unincremented) value to `erase`, but we also increment `i` itself before `erase` begins executing. That’s exactly what we want. As I said, the code is simple, it’s just not what most programmers come up with the first time they try.

Let’s now revise the problem further. Instead of merely erasing each element for which `badValue` returns `true`, we also want to write a message to a log file each time an element is erased.

For the associative containers, this is as easy as easy can be, because it requires only a trivial modification to the loop we just developed:

[Click here to view code image]

```
ofstream logFile;                                   // log
file to write to

AssocContainer<int> c;
...
```

```
for (AssocContainer<int>::iterator i = c.begin();   // loop
conditions are the
     i != c.end(); ) {                              // same as
before

  if (badValue(*i)) {
     logFile << "Erasing " << *i << '\n';           // write
log file
     c.erase(i++);                                  // erase
element
  }
  else ++i;
}
```

It's `vector`, `string`, and `deque` that now give us trouble. We can't use the `erase-remove` idiom any longer, because there's no way to get `erase` or `remove` to write the log file. Furthermore, we can't use the loop we just developed for associative containers, because it yields undefined behavior for `vector`s, `string`s, and `deque`s! Recall that for such containers, invoking `erase` not only invalidates all iterators pointing to the erased element, it also invalidates all iterators *beyond* the erased element. In our case, that includes all iterators beyond `i`. It doesn't matter if we write `i++`, `++i`, or anything else you can think of, because none of the resulting iterators is valid.

We must take a different tack with `vector`, `string`, and `deque`. In particular, we must take advantage of `erase`'s return value. That return value is exactly what we need: it's a valid iterator pointing to the element following the erased element once the `erase` has been accomplished. In other words, we write this:

Click here to view code image

```
for (SeqContainer<int>::iterator i = c.begin();
     i != c.end(); ) {

  if (badValue(*i)) {
     logFile << "Erasing " << *i << '\n';
     i = c.erase(i);                            // keep i
valid by assigning
  }                                             // erase's
return value to it
  else ++i;
}
```

This works wonderfully, but only for the standard sequence containers. Due to reasoning one might question (Item 5 does), `erase`’s return type for the standard associative containers is `void`.[†] For those containers, you have to use the postincrement-the-iterator-you-pass-to-`erase` technique. (Incidentally, this kind of difference between coding for sequence containers and coding for associative containers is an example of why it’s generally ill-advised to try to write container-independent code—see Item 2.)

Lest you be left wondering what the appropriate approach for `list` is, it turns out that for purposes of iterating and erasing, you can treat `list` like a `vector`/`string`/`deque` or you can treat it like an associative container; both approaches work for `list`.

If we take stock of everything we’ve covered in this Item, we come to the following conclusions:

- **To eliminate all objects in a container that have a particular value:**

  If the container is a `vector`, `string`, or `deque`, use the `erase-remove` idiom.

  If the container is a `list`, use `list::remove`.

  If the container is a standard associative container, use its `erase` member function.

- **To eliminate all objects in a container that satisfy a particular predicate:**

  If the container is a `vector`, `string`, or `deque`, use the `erase-remove_if` idiom.

  If the container is a `list`, use `list::remove_if`.

  If the container is a standard associative container, use `remove_copy_if` and `swap`, or write a loop to walk the container elements, being sure to postincrement your iterator when you pass it to `erase`.

- **To do something inside the loop (in addition to erasing objects):**

  If the container is a standard sequence container, write a loop to walk the container elements, being sure to update your iterator with `erase`’s return value each time you call it.

  If the container is a standard associative container, write a loop to walk the container elements, being sure to postincrement your iterator when you pass it to `erase`.

As you can see, there’s more to erasing container elements effectively than just calling `erase`. The best way to approach the matter depends on how you identify which objects to erase, the type of container they're stored in, and what (if anything) you want to do while you're erasing them. As long as you're careful and heed the advice in this Item, you'll have no trouble. If you're not careful, you run the risk of producing code that's needlessly inefficient or that yields undefined behavior.

Item 10: Be aware of allocator conventions and restrictions

Allocators are weird. They were originally developed as an abstraction for memory models that would allow library developers to ignore the distinction between `near` and `far` pointers in certain 16-bit operating systems (i.e., DOS and its pernicious spawn), but that effort failed. Allocators were also designed to facilitate the development of memory managers that are full-fledged objects, but it turned out that that approach led to efficiency degradations in some parts of the STL. To avoid the efficiency hits, the C++ Standardization Committee added wording to the Standard that emasculated allocators as objects, yet simultaneously expressed the hope that they would suffer no loss of potency from the operation.

There's more. Like `operator new` and `operator new[]`, STL allocators are responsible for allocating (and deallocating) raw memory, but an allocator's client interface bears little resemblance to that of `operator new`, `operator new[]`, or even `malloc`. Finally (and perhaps most remarkable), most of the standard containers *never ask* their associated allocator for memory. Never. The end result is that allocators are, well, allocators are weird.

That's not their fault, of course, and at any rate, it doesn't mean they're useless. However, before I explain what allocators are good for (that's the topic of Item 11), I need to explain what they're not good for. There are a number of things that allocators *seem* to be able to do, but can't, and it's important that you know the boundaries of the field before you try to start playing. If you don't, you'll get injured for sure. Besides, the truth about allocators is so peculiar, the mere act of summarizing it is both enlightening and entertaining. At least I hope it is.

The list of restrictions on allocators begins with their vestigial typedefs for pointers and references. As I mentioned, allocators were originally conceived of as abstractions for memory models, and as such it made sense for allocators to provide typedefs for pointers and references in the memory model they defined. In the C++ standard, the default allocator for objects of type `T` (cunningly known as `allocator<T>`) offers the typedefs `allocator<T>::pointer` and `allocator<T>::reference`, and it is expected that user-defined allocators will provide these typedefs, too.

Old C++ hands immediately recognize that this is suspect, because there's no way to fake a reference in C++. Doing so would require the ability to overload `operator.` ("operator dot"), and that's not permitted. In addition, creating objects that act like references is an example of the use of *proxy objects*, and proxy objects lead to a number of problems. (One such problem motivates Item 18. For a comprehensive discussion of proxy objects, turn to Item 30 of *More Effective C++*, where you can read about when they work as well as when they do not.)

In the case of allocators in the STL, it's not any technical shortcomings of proxy objects that render the pointer and reference typedefs impotent, it's the fact that the Standard explicitly allows library implementers to assume that every allocator's `pointer` typedef is a synonym for `T*` and every allocator's `reference` typedef is the same as `T&`. That's right, library implementers may ignore the typedefs and use raw pointers and references directly! So even if you could somehow find a way to write an allocator that successfully provided new pointer and reference types, it wouldn't do any good, because the STL implementations you were using would be free to ignore your typedefs. Neat, huh?

While you're admiring that quirk of standardization, I'll introduce another. Allocators are objects, and that means they may have member functions, nested types and typedefs (such as `pointer` and `reference`), etc., but the Standard says that an implementation of the STL is permitted to assume that all allocator objects of the same type are equivalent and always compare equal. Offhand, that doesn't sound so awful, and there's certainly good motivation for it. Consider this code:

Click here to view code image

```
template<typename T>                          // a user-
defined allocator
class SpecialAllocator { ... };               // template

typedef SpecialAllocator<Widget> SAW;         // SAW =
"SpecialAllocator
```

```
                                                  //         for
Widgets"
list<Widget, SAW> L1;
list<Widget, SAW> L2;

...

L1.splice(L1.begin(), L2);                        // move L2's
nodes to the
                                                  // front of L1
```

Recall that when `list` elements are spliced from one `list` to another, nothing is copied. Instead, a few pointers are adjusted, and the list nodes that used to be in one `list` find themselves in another. This makes splicing operations both fast and exception-safe. In the example above, the nodes that were in `L2` prior to the splice are in `L1` after the splice.

When `L1` is destroyed, of course, it must destroy all its nodes (and deallocate their memory), and because it now contains nodes that were originally part of `L2`, `L1`'s allocator must deallocate the nodes that were originally allocated by `L2`'s allocator. Now it should be clear why the Standard permits implementers of the STL to assume that allocators of the same type are equivalent. It's so memory allocated by one allocator object (such as `L2`'s) may be safely deallocated by another allocator object (such as `L1`'s). Without being able to make such an assumption, splicing operations would be more difficult to implement. Certainly they wouldn't be as efficient as they can be now. (The existence of splicing operations affects other parts of the STL, too. For another example, see Item 4.)

That's all well and good, but the more you think about it, the more you'll realize just how draconian a restriction it is that STL implementations may assume that allocators of the same type are equivalent. It means that portable allocator objects—allocators that will function correctly under different STL implementations—may not have state. Let's be explicit about this: it means that *portable allocators may not have any nonstatic data members*, at least not any that affect their behavior. None. Nada. That means, for example, you can't have one `SpecialAllocator<int>` that allocates from one heap and a different `SpecialAllocator<int>` that allocates from a different heap. Such allocators wouldn't be equivalent, and STL implementations exist where attempts to use both allocators could lead to corrupt runtime data structures.

Notice that this is a *runtime* issue. Allocators with state will compile just fine. They just may not run the way you expect them to. The responsibility for ensuring that all allocators of a given type are equivalent is yours. Don’t expect compilers to issue a warning if you violate this constraint.

In fairness to the Standardization Committee, I should point out that it included the following statement immediately after the text that permits STL implementers to assume that allocators of the same type are equivalent:

> Implementors are encouraged to supply libraries that ... support non-equal instances. In such implementations, ... the semantics of containers and algorithms when allocator instances compare non-equal are implementation-defined.

This is a lovely sentiment, but as a user of the STL who is considering the development of a custom allocator with state, it offers you next to nothing. You can take advantage of this statement only if (1) you know that the STL implementations you are using support inequivalent allocators, (2) you are willing to delve into their documentation to determine whether the implementation-defined behavior of “non-equal” allocators is acceptable to you, and (3) you’re not concerned about porting your code to STL implementations that may take advantage of the latitude expressly extended to them by the Standard. In short, this paragraph—paragraph 5 of section 20.1.5, for those who insist on knowing—is the Standard’s “I have a dream” speech for allocators. Until that dream becomes common reality, programmers concerned about portability will limit themselves to custom allocators with no state.

I remarked earlier that allocators are like `operator new` in that they allocate raw memory, but their interface is different. This becomes apparent if you look at the declaration of the most common forms of `operator new` and `allocator<T>::allocate`:

[Click here to view code image]

```
void* operator new(size_t bytes);

pointer allocator<T>::allocate(size_type numObjects);
                                          // recall that
"pointer" is a typedef
                                          // that's virtually
always T*
```

Both take a parameter specifying how much memory to allocate, but in the case of `operator new`, this parameter specifies a certain number of bytes, while in the case of `allocator<T>::allocate`, it specifies how many `T` objects are to fit in the memory. On a platform where `sizeof(int) == 4`, for example, you’d pass 4 to `operator new` if you wanted enough memory to hold an `int`, but you’d pass 1 to `allocator<int>::allocate`. (The type of this parameter is `size_t` in the case of `operator new`, while it’s `allocator<T>::size_type` in the case of `allocate`. In both cases, it’s an unsigned integral type, and typically `allocator<T>::size_type` is a typedef for `size_t`, anyway.) There’s nothing “wrong” about this discrepancy, but the inconsistent conventions between `operator new` and `allocator<T>::allocate` complicate the process of applying experience with custom versions of `operator new` to the development of custom allocators.

`operator new` and `allocator<T>::allocate` differ in return types, too. `operator new` returns a `void*`, which is the traditional C++ way of representing a pointer to uninitialized memory. `allocator<T>::allocate` returns a `T*` (via the `pointer` typedef), which is not only untraditional, it’s premeditated fraud. The pointer returned from `allocator<T>::allocate` doesn’t point to a `T` object, because no `T` has yet been constructed! Implicit in the STL is the expectation that `allocator<T>::allocate`’s caller will eventually construct one or more `T` objects in the memory it returns (possibly via `allocator<T>::construct`, via `uninitialized_fill`, or via some application of `raw_storage_iterator`s), though in the case of `vector::reserve` or `string::reserve`, that may never happen (see Item 14). The difference in return type between `operator new` and `allocator<T>::allocate` indicates a change in the conceptual model for uninitialized memory, and it again makes it harder to apply knowledge about implementing `operator new` to the development of custom allocators.

That brings us to the final curiosity of STL allocators, that most of the standard containers never make a single call to the allocators with which they are instantiated. Here are two examples:

Click here to view code image

```
list<int> L;                  // same as list<int,
allocator<int> >;
                              // allocator<int> is never
asked to
                              // allocate memory!
```

```
set<Widget, SAW> s;                 // recall that SAW is a
typedef for
                                    //
SpecialAllocator<Widget>; no
                                    // SAW will ever allocate
memory!
```

This oddity is true for `list` and all the standard associative containers (`set`, `multiset`, `map`, and `multimap`). That’s because these are *node-based* containers, i.e., containers based on data structures in which a new node is dynamically allocated each time a value is to be stored. In the case of `list`, the nodes are list nodes. In the case of the standard associative containers, the nodes are usually tree nodes, because the standard associative containers are typically implemented as balanced binary search trees.

Think for a moment about how a `list<T>` is likely to be implemented. The list itself will be made up of nodes, each of which holds a `T` object as well as pointers to the next and previous nodes in the list:

Click here to view code image

```
template<typename T,                                     //
possible list
         typename Allocator = allocator<T> >             //
implementation
class list {
private:

  Allocator alloc;                     // allocator for objects
of type T

  struct ListNode {                    // nodes in the linked
list
    T data;
    ListNode *prev;
    ListNode *next;
  };

  ...

};
```

When a new node is added to the list, we need to get memory for it from an allocator, but we don’t need memory for a `T`, we need memory for a `ListNode` that contains a `T`. That makes our `Allocator` object all but useless, because it doesn’t allocate memory for `ListNode`s, it allocates memory for `T`s. Now you understand why `list` never asks its `Allocator` to do any allocation: the allocator can’t provide what `list` needs.

What `list` needs is a way to get from the allocator type it has to the corresponding allocator for `ListNode`s. This would be tough were it not that, by convention, allocators provide a typedef that does the job. The typedef is called `other`, but it’s not quite that simple, because `other` is a typedef nested inside a struct called `rebind`, which itself is a template nested inside the allocator—which itself is a template!

Please don’t try to think about that last sentence. Instead, look at the code below, then proceed directly to the explanation that follows.

[Click here to view code image]

```
template<typename T>              // the standard allocator
is declared
class allocator {                 // like this, but this
could be a user-
public:                           // written allocator
template, too

  template<typename U>
  struct rebind {
    typedef allocator<U> other;
  };

  ...
};
```

In the code implementing `list<T>`, there is a need to determine the type of the allocator for `ListNode`s that corresponds to the allocator we have for `T`s. The type of the allocator we have for `T`s is the template parameter `Allocator`. That being the case, the type of the corresponding allocator for `ListNode`s is this:

```
Allocator::rebind<ListNode>::other
```

Stay with me here. Every allocator template `A` (e.g., `std::allocator`, `SpecialAllocator`, etc.) is expected to have a nested struct template called `rebind`. `rebind` takes a single type parameter, `U`, and defines nothing but a typedef, `other`. `other` is simply a name for `A<U>`. As a result, `list<T>` can get from its allocator for `T` objects (called `Allocator`) to the corresponding allocator for `ListNode` objects by referring to `Allocator::rebind<ListNode>::other`.

Maybe this makes sense to you, maybe it doesn't. (If you stare at it long enough, it will, but you may have to stare a while. I know I had to.) As a user of the STL who may want to write a custom allocator, you don't really need to know how it works. What you do need to know is that if you choose to write allocators and use them with the standard containers, your allocators must provide the `rebind` template, because standard containers assume it will be there. (For debugging purposes, it's also helpful to know why node-based containers of `T` objects never ask for memory from the allocators for `T` objects.)

Hallelujah! We are finally done examining the idiosyncrasies of allocators. Let us therefore summarize the things you need to remember if you ever want to write a custom allocator.

- Make your allocator a template, with the template parameter `T` representing the type of objects for which you are allocating memory.
- Provide the typedefs `pointer` and `reference`, but always have `pointer` be `T*` and `reference` be `T&`.
- Never give your allocators per-object state. In general, allocators should have no nonstatic data members.
- Remember that an allocator's `allocate` member functions are passed the number of *objects* for which memory is required, not the number of bytes needed. Also remember that these functions return `T*` pointers (via the `pointer` typedef), even though no `T` objects have yet been constructed.
- Be sure to provide the nested `rebind` template on which standard containers depend.

Most of what you have to do to write your own allocator is reproduce a fair amount of boilerplate code, then tinker with a few member functions, notably `allocate` and `deallocate`. Rather than writing the boilerplate from scratch, I suggest you begin with the code at Josuttis' sample allocator web page [23] or in Austern's article, "What Are Allocators Good For?" [24].

Once you've digested the information in this Item, you'll know a lot about what allocators *cannot* do, but that's probably not what you want to know. Instead, you'd probably like to know what allocators *can* do. That's a rich topic in its own right, a topic I call "Item 11."

Item 11: Understand the legitimate uses of custom allocators

So you’ve benchmarked, profiled, and experimented your way to the conclusion that the default STL memory manager (i.e., `allocator<T>`) is too slow, wastes memory, or suffers excessive fragmentation for your STL needs, and you’re certain you can do a better job yourself. Or you discover that `allocator<T>` takes precautions to be thread-safe, but you’re interested only in single-threaded execution and you don’t want to pay for the synchronization overhead you don’t need. Or you know that objects in certain containers are typically used together, so you’d like to place them near one another in a special heap to maximize locality of reference. Or you’d like to set up a unique heap that corresponds to shared memory, then put one or more containers in that memory so they can be shared by other processes. Congratulations! Each of these scenarios corresponds to a situation where custom allocators are well suited to the problem.

For example, suppose you have special routines modeled after `malloc` and `free` for managing a heap of shared memory,

```
void* mallocShared(size_t bytesNeeded);

void freeShared(void *ptr);
```

and you’d like to make it possible to put the contents of STL containers in that shared memory. No problem:

Click here to view code image

```
template<typename T>
class SharedMemoryAllocator {
public:
  ...

  pointer allocate(size_type numObjects, const void
*localityHint = 0)
  {
    return static_cast<pointer>(mallocShared(numObjects *
sizeof(T)));
  }

  void deallocate(pointer ptrToMemory, size_type numObjects)
  {
    freeShared(ptrToMemory);
  }

  ...
};
```

For information on the `pointer` type as well as the cast and the multiplication inside `allocate`, see Item 10.

You could use `SharedMemoryAllocator` like this:

Click here to view code image

```
// convenience typedef
typedef
  vector<double, SharedMemoryAllocator<double> >
SharedDoubleVec;
...

{                                    // begin some block
  ...

  SharedDoubleVec v;                 // create a vector whose
elements
                                     // are in shared memory
  ...
}                                    // end the block
```

The wording in the comment next to `v`'s definition is important. `v` is using a `SharedMemoryAllocator`, so the memory `v` allocates to hold its elements will come from shared memory. `v` itself, however—including all its data members—will almost certainly *not* be placed in shared memory. `v` is just a normal stack-based object, so it will be located in whatever memory the runtime system uses for all normal stack-based objects. That's almost never shared memory. To put both `v`'s contents and `v` itself into shared memory, you'd have to do something like this:

Click here to view code image

```
void *pVectorMemory =                       // allocate enough
shared
  mallocShared(sizeof(SharedDoubleVec));    // memory to hold a
                                            // SharedDoubleVec
object

SharedDoubleVec *pv =                       // use "placement
new" to
```

```
  new (pVectorMemory) SharedDoubleVec;     // create a
SharedDoubleVec
                                           // object in the
memory;
                                           // see below

...                                        // use the object
(via pv)

pv->~SharedDoubleVec();                    // destroy the
object in the
                                           // shared memory

freeShared(pVectorMemory);                 // deallocate the
initial
                                           // chunk of shared
memory
```

I hope the comments make clear how this works. Fundamentally, you acquire some shared memory, then construct a `vector` in it that uses shared memory for its own internal allocations. When you're done with the vector, you invoke its destructor, then release the memory the vector occupied. The code isn't terribly complicated, but it's a lot more demanding than just declaring a local variable as we did above. Unless you really need a container (as opposed to its elements) to be in shared memory, I encourage you to avoid this manual four-step allocate/construct/destroy/deallocate process.

In this example, you've doubtless noticed that the code ignores the possibility that `mallocShared` might return a null pointer. Obviously, production code would have to take such a possibility into account. Also, construction of the `vector` in the shared memory is accomplished by "placement `new`." If you're unfamiliar with placement `new`, your favorite C++ text should be able to introduce you. If that text happens to be *More Effective C++*, you'll find that the pleasantries are exchanged in Item 8.

As a second example of the utility of allocators, suppose you have two heaps, identified by the classes `Heap1` and `Heap2`. Each heap class has static member functions for performing allocation and deallocation:

Click here to view code image

```
class Heap1 {
public:
  ...
  static void* alloc(size_t numBytes, const void
*memoryBlockToBeNear);
  static void dealloc(void *ptr);
  ...
};

class Heap2 { ... };                    // has the same
alloc/dealloc interface
```

Further suppose you'd like to co-locate the contents of some STL containers in different heaps. Again, no problem. First you write an allocator designed to use classes like `Heap1` and `Heap2` for the actual memory management:

Click here to view code image

```
template<typename T, typename Heap>
class SpecificHeapAllocator {
public:
  ...

  pointer allocate(size_type numObjects, const void
*localityHint = 0)
  {
    return static_cast<pointer> (Heap::alloc(numObjects *
sizeof(T),
                                    localityHint));
}

void deallocate(pointer ptrToMemory, size_type numObjects)
{
   Heap::dealloc(ptrToMemory);
}

...
};
```

Then you use `SpecificHeapAllocator` to cluster containers' elements together:

Click here to view code image

```
vector<int, SpecificHeapAllocator<int, Heap1> > v;        //
put both v's and
set<int, SpecificHeapAllocator<int, Heap1> > s;           //
s's elements in
                                                          //
Heap1

list<Widget,
     SpecificHeapAllocator<Widget, Heap2> > L;            //
put both L's and
map<int, string, less<int>,                               //
m's elements in
    SpecificHeapAllocator<pair<const int, string>,        //
Heap2
                          Heap2> > m;
```

In this example, it’s quite important that `Heap1` and `Heap2` be *types* and not objects. The STL offers a syntax for initializing different STL containers with different allocator objects of the same type, but I’m not going to show you what it is. That’s because if `Heap1` and `Heap2` were *objects* instead of types, they’d be inequivalent allocators, and that would violate the equivalence constraint on allocators that is detailed in Item 10.

As these examples demonstrate, allocators are useful in a number of contexts. As long as you obey the constraint that all allocators of the same type must be equivalent, you’ll have no trouble employing custom allocators to control general memory management strategies, clustering relationships, and use of shared memory and other special heaps.

Item 12: Have realistic expectations about the thread safety of STL containers

The world of standard C++ is rather sheltered and old-fashioned. In this rarefied world, all executables are statically linked. Neither memory-mapped files nor shared memory exist. There are no window systems, no networks, no databases, no other processes. That being the case, you should not be surprised to learn that the Standard says not a word about threading. The first expectation you should have about the thread safety of the STL, then, is that it will vary from implementation to implementation.

Of course, multithreaded programs are common, so most STL vendors strive to make their implementations work well in a threaded environment. Even when they do a good job, however, much of the burden remains on your shoulders, and it’s important to understand why. There’s only so much STL vendors can do to ease your multithreading pain, and you need to know what it is.

The gold standard in support for multithreading in STL containers (and the aspiration of most vendors) has been defined by SGI and is published at their

STL Web Site [21]. In essence, it says that the most you can hope for from an implementation is the following.

- **Multiple readers are safe.** Multiple threads may simultaneously read the contents of a single container, and this will work correctly. Naturally, there must not be any writers acting on the container during the reads.
- **Multiple writers to different containers are safe.** Multiple threads may simultaneously write to different containers.

That's all, and let me make clear that this is what you can *hope for*, not what you can *expect*. Some implementations offer these guarantees, but some do not.

Writing multithreaded code is hard, and many programmers wish that STL implementations were completely thread safe out of the box. Were that the case, programmers might hope to be relieved of the need to attend to concurrency control themselves. There's no doubt this would be a convenient state of affairs, but it would also be very difficult to achieve. Consider the following ways a library might try to implement such comprehensive container thread safety:

- Lock a container for the duration of each call to its member functions.
- Lock a container for the lifetime of each iterator it returns (via, e.g., calls to `begin` or `end`).
- Lock a container for the duration of each algorithm invoked on that container. (This actually makes no sense, because, as Item 32 explains, algorithms have no way to identify the container on which they are operating. Nevertheless, we'll examine this option here, because it's instructive to see why it wouldn't work even if it were possible.)

Now consider the following code. It searches a `vector<int>` for the first occurrence of the value 5, and, if it finds one, changes that value to 0.

Click here to view code image

```
vector<int> v;

...

vector<int>::iterator first5(find(v.begin(), v.end(),
5));      // Line 1
```

```
if (first5 != v.end())
{                                         // Line 2
    *first5 =
0;                                                 // Line 3
}
```

In a multithreaded environment, it’s possible that a different thread will modify the data in `v` immediately after completion of Line 1. If that were to happen, the test of `first5` against `v.end` on Line 2 would be meaningless, because `v`’s values would be different from what they were at the end of Line 1. In fact, such a test could yield undefined results, because another thread could have intervened between Lines 1 and 2 and invalidated `first5`, perhaps by performing an insertion that caused the vector to reallocate its underlying memory. (That would invalidate all the vector’s iterators. For details on this reallocation behavior, turn to Item 14.) Similarly, the assignment to `*first5` on Line 3 is unsafe, because another thread might execute between Lines 2 and 3 in such a way as to invalidate `first5`, perhaps by erasing the element it points to (or at least *used* to point to).

None of the approaches to locking listed above would prevent these problems. The calls to `begin` and `end` in Line 1 both return too quickly to offer any help, the iterators they generate last only until the end of that line, and `find` also returns at the end of that line.

For the code above to be thread safe, `v` must remain locked from Line 1 through Line 3, and it’s difficult to imagine how an STL implementation could deduce that automatically. Bearing in mind the typically high cost of synchronization primitives (e.g., semaphores, mutexes, etc.), it’s even more difficult to imagine how an implementation could do it without imposing a significant performance penalty on programs that knew *a priori*—that were *designed* in such a way—that no more than one thread had access to `v` during the course of Lines 1-3.

Such considerations explain why you can’t expect any STL implementation to make your threading woes disappear. Instead, you’ll have to manually take charge of synchronization control in these kinds of scenarios. In this example, you might do it like this:

Click here to view code image

```
vector<int> v;
```

```
...

getMutexFor(v);

vector<int>::iterator first5(find(v.begin(), v.end(), 5));
if (first5 != v.end())
{                                     // this is now safe
    *first5 =
0;                                             // so is this
}

releaseMutexFor(v);
```

A more object-oriented solution is to create a `Lock` class that acquires a mutex in its constructor and releases it in its destructor, thus minimizing the chances that a call to `getMutexFor` will go unmatched by a call to `releaseMutexFor`. The essence of such a class (really a class template) is this:

Click here to view code image

```
template<typename Container>       // skeletal template for
classes
class Lock {                       // that acquire and
release mutexes
public:                            // for containers; many
details
                                   // have been omitted
  Lock(const Container& container)
  : c(container)
  {
      getMutexFor(c);              // acquire mutex in the
constructor
  }

  ~Lock()
  {
      releaseMutexFor(c);          // release it in the
destructor
  }
private:
  const Container& c;
};
```

The idea of using a class (like `Lock`) to manage the lifetime of resources (such as mutexes) is generally known as *resource acquisition is initialization*, and you should be able to read about it in any comprehensive C++ textbook. A good place to start is Stroustrup's *The C++ Programming Language* [7], because Stroustrup popularized the idiom, but you can also turn to Item 9 of *More Effective C++*. No matter what source you consult, bear in mind that the above `Lock` is stripped to the bare essentials. An industrial-strength version would require a number of enhancements, but such a fleshing-out would have nothing to do with the STL. Furthermore, this minimalist `Lock` is enough to see how we could apply it to the example we've been considering:

Click here to view code image

```
vector<int> v;

...

{                                                  // create
new block
  Lock<vector<int> > lock(v);                      // acquire
mutex

  vector<int>::iterator first5(find(v.begin(), v.end(), 5));

  if (first5 != v.end()) {
      *first5 = 0;
  }
}                                                  // close
block, automatically
                                                   //
releasing the mutex
```

Because a `Lock` object releases the container's mutex in the `Lock`'s destructor, it's important that the `Lock` be destroyed as soon as the mutex should be released. To make that happen, we create a new block in which to define the `Lock`, and we close that block as soon as we no longer need the mutex. This sounds like we're just trading the need to call `releaseMutexFor` with the need to close a new block, but that's not an accurate assessment. If we forget to create a new block for the `Lock`, the mutex will still be released, but it may happen later than it should—when control reaches the end of the enclosing block. If we forget to call `releaseMutexFor`, we never release the mutex.

Furthermore, the `Lock`-based approach is robust in the presence of exceptions. C++ guarantees that local objects are destroyed if an exception is thrown, so `Lock` will release its mutex even if an exception is thrown while we're using the `Lock` object.[†] If we relied on manual calls to `getMutexFor` and `releaseMutexFor`, we’d never relinquish the mutex if an exception was thrown after calling `getMutexFor` but before calling `releaseMutexFor`.

Exceptions and resource management are important, but they’re not the subject of this Item. This Item is about thread safety in the STL. When it comes to thread safety and STL containers, you can hope for a library implementation that allows multiple readers on one container and multiple writers on separate containers. You *can’t* hope for the library to eliminate the need for manual concurrency control, and you can’t *rely* on any thread support at all.

Chapter 2. `vector` and `string`

All the STL containers are useful, but if you're like most C++ programmers, you'll find yourself reaching for `vector` and `string` more often than their compatriots. That's to be expected. `vector` and `string` are designed to replace most applications of arrays, and arrays are so useful, they've been included in every commercially successful programming language from COBOL to Java.

The Items in this chapter cover `vector`s and `string`s from a number of perspectives. We begin with a discussion of why the switch from arrays is worthwhile, then look at ways to improve `vector` and `string` performance, identify important variations in `string` implementations, examine how to pass `vector` and `string` data to APIs that understand only C, and learn how to eliminate excess memory allocation. We conclude with an examination of an instructive anomaly, `vector<bool>`, the little vector that couldn't.

Each of the Items in this chapter will help you take the two most useful containers in the STL and refine their application. By the time we're done, you'll know how to make them serve you even better.

Item 13: Prefer `vector` and `string` to dynamically allocated arrays

The minute you decide to use `new` for a dynamic allocation, you adopt the following responsibilities:

1. You must make sure that somebody will later `delete` the allocation. Without a subsequent `delete`, your `new` will yield a resource leak.
2. You must ensure that the correct form of `delete` is used. For an allocation of a single object, "`delete`" must be used. For an array allocation, "`delete []`" is required. If the wrong form of `delete` is used, results will be undefined. On some platforms, the program will crash at runtime. On others, it will silently blunder forward, sometimes leaking resources and corrupting memory as it goes.
3. You must make sure that `delete` is used exactly once. If an allocation is deleted more than once, results are again undefined.

That's quite a set of responsibilities, and I can't understand why you'd want to adopt them if it wasn't necessary. Thanks to `vector` and `string`, it isn't necessary anywhere near as often as it used to be.

Any time you find yourself getting ready to dynamically allocate an array (i.e., plotting to write "`new T[...]`"), you should consider using a `vector` or a `string` instead. (In general, use `string` when `T` is a character type and use `vector` when it's not, though later in this Item, we'll encounter a scenario where a `vector<char>` may be a reasonable design choice.) `vector` and `string` eliminate the burdens above, because they manage their own memory. Their memory grows as elements are added to these containers, and when a `vector` or `string` is destroyed, its destructor automatically destroys the elements in the container and deallocates the memory holding those elements.

In addition, `vector` and `string` are full-fledged STL sequence containers, so they put at your disposal the complete arsenal of STL algorithms that work on such containers. True, arrays can be used with STL algorithms, too, but arrays don't offer member functions like `begin`, `end`, and `size`, nor do they have nested typedefs like `iterator`, `reverse_iterator`, or `value_type`. And of course `char*` pointers can hardly compete with the scores of specialized member functions proffered by `string`. The more you work with the STL, the more jaundiced the eye with which you'll come to view built-in arrays.

If you're concerned about the legacy code you must continue to support, all of which is based on arrays, relax and use `vector`s and `string`s anyway. Item 16 shows how easy it is to pass the data in `vector`s and `string`s to APIs that expect arrays, so integration with legacy code is generally not a problem.

Frankly, I can think of only one legitimate cause for concern in replacing dynamically allocated arrays with `vector`s or `string`s, and it applies only to `string`s. Many `string` implementations employ reference counting behind the scenes (see Item 15), a strategy that eliminates some unnecessary memory allocations and copying of characters and that can improve performance for many applications. In fact, the ability to optimize `string`s via reference counting was considered so important, the C++ Standardization Committee took specific steps to make sure it was a valid implementation.

Alas, one programmer's optimization is another's pessimization, and if you use reference-counted `string`s in a multithreaded environment, you may find that the time saved by avoiding allocations and copying is dwarfed by the time spent on behind-the-scenes concurrency control. (For details, consult Sutter’s article, “Optimizations That Aren’t (In a Multithreaded World)” [20].) If you’re using reference-counted strings in a multithreaded environment, then, it makes sense to keep an eye out for performance problems arising from their support for thread safety.

To determine whether you’re using a reference-counting implementation for `string`, it’s often easiest to consult the documentation for your library. Because reference counting is considered an optimization, vendors generally tout it as a feature. An alternative is to look at the source code for your libraries’ implementations of `string`. I don’t generally recommend trying to figure things out from library source code, but sometimes it’s the only way to find out what you need to know. If you choose this approach, remember that `string` is a typedef for `basic_string<char>` (and `wstring` is a typedef for `basic_string<wchar_t>`), so what you really want to look at is the template `basic_string`. The easiest thing to check is probably the class’s copy constructor. Look to see if it increments a reference count somewhere. If it does, `string` is reference counted. If it doesn’t, either `string` isn’t reference counted or you misread the code. Ahem.

If the `string` implementations available to you are reference counted and you are running in a multithreaded environment where you’ve determined that `string`’s reference counting support is a performance problem, you have at least three reasonable choices, none of which involves abandoning the STL. First, check to see if your library implementation is one that makes it possible to disable reference counting, often by changing the value of a preprocessor variable. This won’t be portable, of course, but given the amount of work involved, it’s worth investigating. Second, find or develop an alternative `string` implementation (or partial implementation) that doesn’t use reference counting. Third, consider using a `vector<char>` instead of a `string`. `vector` implementations are not allowed to be reference counted, so hidden multithreading performance issues fail to arise. Of course, you forgo `string`’s fancy member functions if you switch to `vector<char>`, but most of that functionality is available through STL algorithms anyway, so you’re not so much giving up functionality as you are trading one syntax for another.

The upshot of all this is simple. If you’re dynamically allocating arrays, you’re probably taking on more work than you need to. To lighten your load, use `vector`s or `string`s instead.

Item 14: Use `reserve` to avoid unnecessary reallocations

One of the most marvelous things about STL containers is that they automatically grow to accommodate as much data as you put into them, provided only that you don't exceed their maximum size. (To discover this maximum, just call the aptly named `max_size` member function.) For `vector` and `string`, growth is handled by doing the moral equivalent of a `realloc` whenever more space is needed. This `realloc`-like operation has four parts:

1. Allocate a new block of memory that is some multiple of the container's current capacity. In most implementations, `vector` and `string` capacities grow by a factor of between 1.5 and 2 each time.
2. Copy all the elements from the container's old memory into its new memory.
3. Destroy the objects in the old memory.
4. Deallocate the old memory.

Given all that allocation, deallocation, copying, and destruction, it should not stun you to learn that these steps can be expensive. Naturally, you don't want to perform them any more frequently than you have to. If that doesn't strike you as natural, perhaps it will when you consider that each time these steps occur, all iterators, pointers, and references into the `vector` or `string` are invalidated. That means that the simple act of inserting an element into a `vector` or `string` may also require updating other data structures that use iterators, pointers, or references into the `vector` or `string` being expanded.

The `reserve` member function allows you to minimize the number of reallocations that must be performed, thus avoiding the costs of reallocation and iterator/pointer/reference invalidation. Before I explain how `reserve` can do that, however, let me briefly recap four interrelated member functions that are sometimes confused. Among the standard containers, only `vector` and `string` offer all of these functions.

- `size()` tells you how many elements are in the container. It does *not* tell you how much memory the container has allocated for the elements it holds.
- `capacity()` tells you how many elements the container can hold in the memory it has already allocated. This is how many *total elements* the container can hold in that memory, not how many *more* elements it can

hold. If you'd like to find out how much unoccupied memory a `vector` or `string` has, you must subtract `size()` from `capacity()`. If `size` and `capacity` return the same value, there is no empty space in the container, and the next insertion (via `insert` or `push_back`, etc.) will trigger the reallocation steps above.

- `resize(`*`Container`*`::size_type n)` forces the container to change to `n` the number of elements it holds. After the call to `resize`, `size` will return `n`. If `n` is smaller than the current size, elements at the end of the container will be destroyed. If `n` is larger than the current size, new elements will be added to the end of the container, either by copying a default-constructed element or by copying an object you provide via an additional parameter. If `n` is larger than the current capacity, a reallocation will take place before elements are added.
- `reserve`(*`Container`*`::size_type n)` forces the container to change its capacity to at least `n`, provided `n` is no less than the current size. This typically forces a reallocation, because the capacity needs to be increased. (If `n` is less than the current capacity, `vector` ignores the call and does nothing. `string` *may* reduce its capacity to the maximum of `size()` and `n`, but the `string`'s size definitely remains unchanged. In my experience, using `reserve` to trim the excess capacity from a `string` is generally less successful than using "the swap trick," which is the topic of Item 17.) Note that calling `reserve` never changes the number of elements in a container (i.e., its size).

This recap should make it clear that reallocations (including their constituent memory allocations and deallocations, object copying and destruction, and invalidation of iterators, pointers, and references) occur whenever an element needs to be inserted and the container's capacity is insufficient. The key to avoiding reallocations, then, is to use `reserve` to set a container's capacity to a sufficiently large value as soon as possible, ideally right after the container is constructed.

For example, suppose you'd like to create a `vector<int>` holding the values 1–1000. Without using `reserve`, you might do it like this:

```
vector<int> v;

for (int i = 1; i <= 1000; ++i) v.push_back(i);
```

This code will typically result in between two and 18 reallocations during the course of the loop.

Modifying the code to use `reserve` gives us this:

Click here to view code image

```
vector<int> v;
v.reserve(1000);

for (int i = 1; i <= 1000; ++i) v.push_back(i);
```

This should result in zero reallocations during the loop.

The relationship between `size` and `capacity` makes it possible to predict when an insertion will cause a `vector` or `string` to perform a reallocation, and that, in turn, makes it possible to predict when an insertion will invalidate iterators, pointers, and references into a container. For example, given this code,

```
string s;

...

if (s.size() < s.capacity()) {
    s.push_back('x');
}
```

the call to `push_back` can't invalidate iterators (other than to `s`'s end), pointers, or references into the string, because the string's capacity is guaranteed to be greater than its size. If, instead of performing a `push_back`, the code did an `insert` into an arbitrary location in the string, we'd still be guaranteed that no reallocation would take place during the insertion, but, in accord with the usual rule for iterator invalidation accompanying a `string` insertion, all iterators/pointers/references from the insertion point to the end of the string would be invalidated.

Getting back to the main point of this Item, there are two common ways to use `reserve` to avoid unneeded reallocations. The first is applicable when you know exactly or approximately how many elements will ultimately end up in your container. In that case, as in the `vector` code above, you simply reserve the appropriate amount of space in advance. The second way is to reserve the maximum space you could ever need, then, once you've added all your data, trim off any excess capacity. The trimming part isn't difficult, but I'm not going to show it here, because there's a trick to it. To learn the trick, turn to Item 17.

Item 15: Be aware of variations in `string` implementations

Bjarne Stroustrup once wrote an article with the curious title, “Sixteen Ways to Stack a Cat” [27]. It turns out that there are almost as many ways to implement `string`s. As experienced and sophisticated software engineers, of course, we’re supposed to pooh-pooh “implementation details,” but if Einstein is right and God is in the details, reality requires that we sometimes get religion. Even when the details don’t matter, having some idea about them puts us in a position to be *sure* they don’t matter.

For example, what is the size of a `string` object? In other words, what value does `sizeof(string)` return? This could be an important question if you’re keeping a close eye on memory consumption, and you’re thinking of replacing a raw `char*` pointer with a `string` object.

The news about `sizeof(string)` is “interesting,” which is almost certainly what you do not want to hear if you’re concerned about space. While it’s not uncommon to find `string` implementations in which `string`s are the same size as `char*` pointers, it’s also easy to find `string` implementations where each `string` is seven times that size. Why the difference? To understand that, we have to know what data a `string` is likely to store as well as where it might decide to store it.

Virtually every `string` implementation holds the following information:

- The **size** of the string, i.e., the number of characters it contains.
- The **capacity** of the memory holding the string’s characters. (For a review of the difference between a string’s size and its capacity, see Item 14.)
- The **value** of the string, i.e., the characters making up the string.

In addition, a `string` may hold

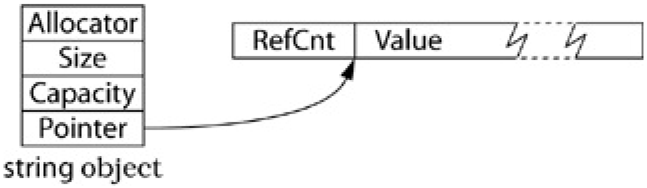
- A copy of its **allocator**. For an explanation of why this field is optional, turn to Item 10 and read about the curious rules governing allocators.

`string` implementations that depend on reference counting also contain

- The **reference count** for the value.

Different `string` implementations put these pieces of information together in different ways. To demonstrate what I mean, I’ll show you the data structures used by four different `string` implementations. There’s nothing special about these selections. They all come from STL implementations that are commonly used. They just happen to be the `string` implementations in the first four libraries I checked.

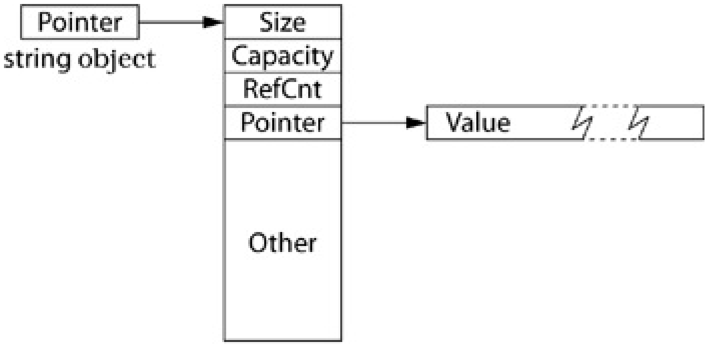
In implementation A, each `string` object contains a copy of its allocator, the string's size, its capacity, and a pointer to a dynamically allocated buffer containing both the reference count (the "`RefCnt`") and the string's value. In this implementation, a `string` object using the default allocator is four times the size of a pointer. With a custom allocator, the `string` object would be bigger by about the size of the allocator object:



Implementation A

Implementation B's `string` objects are the same size as a pointer, because they contain nothing but a pointer to a struct. Again, this assumes that the default allocator is used. As in Implementation A, if a custom allocator is used, the `string` object's size will increase by about the size of the allocator object. In this implementation, use of the default allocator requires no space, thanks to an optimization present here but not present in Implementation A.

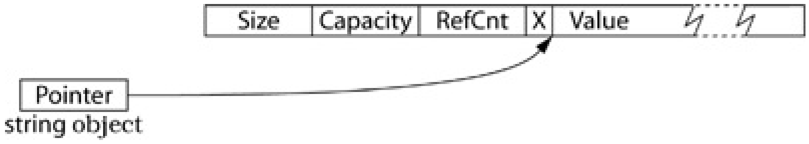
The object pointed to by B's `string` contains the string's size, capacity, and reference count, as well as a pointer to a dynamically allocated buffer holding the string's value. The object also contains some additional data related to concurrency control in multithreaded systems. Such data is outside our purview here, so I've just labeled that part of the data structure "Other:"



Implementation B

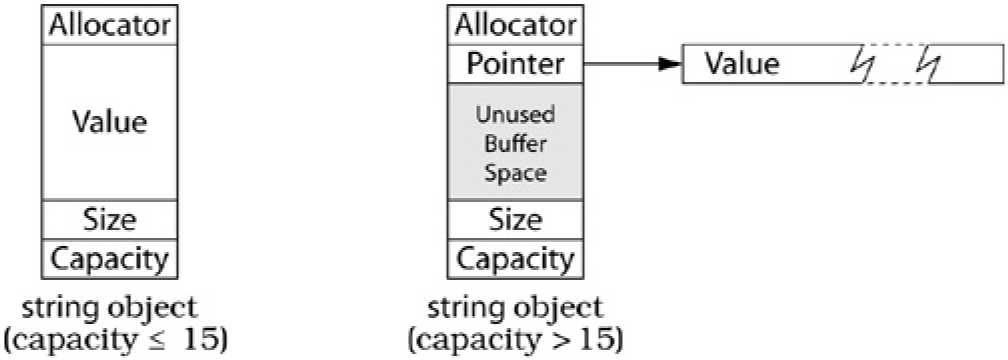
The box for “Other” is larger than the other boxes, because I’ve drawn the boxes to scale. If one box is twice the size of another, the larger box uses twice as many bytes as the smaller one. In Implementation B, the data for concurrency control is six times the size of a pointer.

`string` objects under Implementation C are always the size of a pointer, but this pointer points to a dynamically allocated buffer containing everything related to the string: its size, capacity, reference count, and value. There is no per-object allocator support. The buffer also holds some data concerning the shareability of the value, a topic we’ll not consider here, so I’ve labeled it “X”. (If you’re interested in why a reference counted value might not be shareable in the first place, consult Item 29 of *More Effective C++*.)


Size
Capacity
RefCnt
X
Value
Pointer
string object

Implementation C

Implementation D’s `string` objects are seven times the size of a pointer (still assuming use of the default allocator). This implementation employs no reference counting, but each `string` contains an internal buffer large enough to represent string values of up to 15 characters. Small strings can thus be stored entirely within the `string` object, a feature sometimes known as the “small string optimization.” When a `string`’s capacity exceeds 15, the first part of the buffer is used as a pointer to dynamically allocated memory, and the `string`’s value resides in that memory:


Allocator
Value
Size
Capacity
string object
(capacity ≤ 15)
Allocator
Pointer
Value
Unused
Buffer
Space
Size
Capacity
string object
(capacity > 15)

Implementation D

These diagrams do more than just prove I can read source code and draw pretty pictures. They also allow you to deduce that creation of a `string` in a statement such as this,

Click here to view code image

```
string s("Perse");         // Our dog is named "Persephone",
but we
                           // usually just call her "Perse".
Visit her web site
                           // at
http://www.aristeia.com/Persephone/
```

will cost you zero dynamic allocations under Implementation D, one under Implementations A and C, and two under Implementation B (one for the object pointed to by the `string` object, one for the character buffer pointed to by that object). If you're concerned about the number of times you dynamically allocate and deallocate memory, or if you're concerned about the memory overhead that often accompanies such allocations, you might want to shy away from Implementation B. On the other hand, the fact that Implementation B's data structure includes specific support for concurrency control in multithreaded systems might mean that it meets your needs better than implementations A or C, the number of dynamic allocations notwithstanding. (Implementation D requires no special support for multithreading, because it doesn't use reference counting. For more on the interaction of threading and reference counted strings, see Item 13. For information on what you may reasonably hope for in the way of threading support in STL containers, consult Item 12.)

In a design based on reference counting, everything outside the `string` object can be shared by multiple `string`s (if they have the same value), so something else we can observe from the diagrams is that Implementation A offers less sharing than B or C. In particular, Implementations B and C can share a `string`'s size and capacity, thus potentially reducing the amortized per-object cost of storing that data. Interestingly, the fact that Implementation C fails to support per-object allocators means that it is the only implementation that can share allocators: all `string`s must use the same one! (Again, for details on the rules governing allocators, turn to Item 10.) Implementation D shares no data across `string` objects.

An interesting aspect of `string` behavior you can't fully deduce from the diagrams is the policy regarding memory allocation for small strings. Some implementations refuse to allocate memory for fewer than a certain number of characters, and Implementations A and D are among them. Look again at this statement:

```
string s("Perse");                      // s is a string of
size 5
```

Implementation A has a minimum allocation size of 32 characters, so though `s`'s size is 5 under all implementations, its capacity under Implementation A is 31. (The 32nd character is presumably reserved for a trailing null, thus making it easy to implement the `c_str` member function.) Implementation D's minimum buffer size is 16, including room for a trailing null, so under Implementation D, `s`'s capacity is 15. Of course, what distinguishes Implementation D in this area is that the memory for `string`s with capacity less than 16 is contained within the `string` object itself. Implementation B has no minimum allocation, and under Implementation B, `s`'s capacity is 7. (Why it's not 6 or 5, I don't know. I didn't read the source code that closely, sorry.)

I hope it's obvious that the various implementations' policies on minimum allocations might be important to you if you expect to have lots of short strings and either (1) your release environment has very little memory or (2) you are concerned about locality of reference and want to cluster strings on as few pages as possible.

Clearly, `string` implementations have more degrees of freedom than are apparent at first glance, and equally clearly, different implementers have taken advantage of this design flexibility in different ways. Let's summarize the things that vary:

- `string` values may or may not be reference counted. By default, many implementations do use reference counting, but they usually offer a way to turn it off, often via a preprocessor macro. Item 13 gives an example of specific conditions under which you might want to turn it off, but you might want to do so for other reasons, too. For example, reference counting helps only when strings are frequently copied, and some applications just don't copy strings often enough to justify the overhead.
- `string` objects may range in size from one to at least seven times the size of `char*` pointers.
- Creation of a new string value may require zero, one, or two dynamic allocations.
- `string` objects may or may not share information on the string's size and

capacity.

- `string`s may or may not support per-object allocators.
- Different implementations have different policies regarding minimum allocations for character buffers.

Now, don't get me wrong. I think `string` is one of the most important components of the standard library, and I encourage you to use it as often as you can. Item 13, for example, is devoted to why you should use `string` in place of dynamically allocated character arrays. At the same time, if you're to make effective use of the STL, you need to be aware of the variability in `string` implementations, especially if you're writing code that must run on different STL platforms and you face stringent performance requirements.

Besides, `string` seems so conceptually simple. Who'd have thought the implementations could be so interesting?

Item 16: Know how to pass `vector` and `string` data to legacy APIs

Since C++ was standardized in 1998, the C++ elite haven't been terribly subtle in their attempt to nudge programmers away from arrays and towards `vector`s. They've been similarly overt in trying to get developers to shift from `char*` pointers to `string` objects. There are good reasons for making these changes, including the elimination of common programming errors (see Item 13) and the ability to take full advantage of the power of the STL algorithms (see, e.g., Item 31).

Still, obstacles remain, and one of the most common is the existence of legacy C APIs that traffic in arrays and `char*` pointers instead of `vector` and `string` objects. Such APIs will exist for a long time, so we must make peace with them if we are to use the STL effectively.

Fortunately, it's easy. If you have a `vector v` and you need to get a pointer to the data in `v` that can be viewed as an array, just use `&v[0]`. For a `string s`, the corresponding incantation is simply `s.c_str()`. But read on. As the fine print in advertising often points out, certain restrictions apply.

Given

```
vector<int> v;
```

the expression `v[0]` yields a reference to the first element in the vector, so `&v[0]` is a pointer to that first element. The elements in a `vector` are constrained by the C++ Standard to be stored in contiguous memory, just like an array, so if we wish to pass `v` to a C API that looks something like this,

```
void doSomething(const int* pInts, size_t numInts);
```

we can do it like this:

```
doSomething(&v[0], v.size());
```

Maybe. Probably. The only sticking point is if `v` is empty. If it is, `v.size()` is zero, and `&v[0]` attempts to produce a pointer to something that does not exist. Not good. Undefined results. A safer way to code the call is this:

```
if (!v.empty()) {
    doSomething(&v[0], v.size());
}
```

If you travel in the wrong circles, you may run across shady characters who will tell you that you can use `v.begin()` in place of `&v[0]`, because (these loathsome creatures will tell you) `begin` returns an iterator into the `vector`, and for `vector`s, iterators are really pointers. That’s often true, but as Item 50 reveals, it’s not always true, and you should never rely on it. The return type of `begin` is an iterator, not a pointer, and you should never use `begin` when you need to get a pointer to the data in a `vector`. If you’re determined to type `v.begin()` for some reason, type `&*v.begin()`, because that will yield the same pointer as `&v[0]`, though it’s more work for you as a typist and more obscure for people trying to make sense of your code. Frankly, if you’re hanging out with people who tell you to use `v.begin()` instead of `&v[0]`, you need to rethink your social circle.

The approach to getting a pointer to container data that works for `vector`s isn’t reliable for `string`s, because (1) the data for `string`s are not guaranteed to be stored in a single chunk of contiguous memory, and (2) the internal representation of a `string` is not guaranteed to end with a null character. This explains the existence of the `string` member function `c_str`, which returns a pointer to the value of the string in a form designed for C. We can thus pass a string `s` to this function,

```
void doSomething(const char *pString);
```

like this:

```
doSomething(s.c_str());
```

This works even if the string is of length zero. In that case, `c_str` will return a pointer to a null character. It also works if the string has embedded nulls. If it does, however, `doSomething` is likely to interpret the first embedded null as the end of the string. `string` objects don't care if they contain null characters, but `char*`-based C APIs do.

Look again at the `doSomething` declarations:

```
void doSomething(const int* pInts, size_t numInts);

void doSomething(const char *pString);
```

In both cases, the pointers being passed are pointers to `const`. The `vector` or `string` data are being passed to an API that will *read* it, not modify it. This is by far the safest thing to do. For `string`s, it's the only thing to do, because there is no guarantee that `c_str` yields a pointer to the internal representation of the string data; it could return a pointer to an unmodifiable *copy* of the string's data, one that's correctly formatted for a C API. (If this makes the efficiency hairs on the back of your neck rise up in alarm, rest assured that the alarm is probably false. I don't know of any contemporary library implementation that takes advantage of this latitude.)

For a `vector`, you have a little more flexibility. If you pass `v` to a C API that modifies `v`'s elements, that's typically okay, but the called routine must not attempt to change the number of elements in the vector. For example, it must not try to "create" new elements in a `vector`'s unused capacity. If it does, `v` will become internally inconsistent, because it won't know its correct size any longer. `v.size()` will yield incorrect results. And if the called routine attempts to add data to a vector whose size and capacity (see Item 14) are the same, truly horrible things could happen. I don't even want to contemplate them. They're just too awful.

Did you notice my use of the word "typically" in the phrase "that's typically okay" in the preceding paragraph? Of course you did. Some vectors have extra constraints on their data, and if you pass a vector to an API that modifies the vector's data, you must ensure that the additional constraints continue to be satisfied. For example, Item 23 explains how sorted vectors can often be a viable alternative to associative containers, but it's important for such vectors to remain sorted. If you pass a sorted vector to an API that may modify the vector's data, you'll need to take into account that the vector may no longer be sorted after the call has returned.

If you have a `vector` that you'd like to initialize with elements from a C API, you can take advantage of the underlying layout compatibility of `vector`s and arrays by passing to the API the storage for the vector's elements:

Click here to view code image

```
// C API: this function takes a pointer to an array of at
most arraySize
// doubles and writes data to it. It returns the number of
doubles written,
// which is never more than arraySize.
size_t fillArray(double *pArray, size_t arraySize);

vector<double> vd(maxNumDoubles);                  // create
a vector whose
                                                   // size is
maxNumDoubles

vd.resize(fillArray(&vd[0], vd.size()));           // have
fillArray write data
                                                   // into
vd, then resize vd
                                                   // to the
number of
                                                   //
elements fillArray wrote
```

This technique works only for `vector`s, because only `vector`s are guaranteed to have the same underlying memory layout as arrays. If you want to initialize a `string` with data from a C API, however, you can do it easily enough. Just have the API put the data into a `vector<char>`, then copy the data from the vector to the string:

Click here to view code image

```
// C API: this function takes a pointer to an array of at
most arraySize
// chars and writes data to it. It returns the number of
chars written,
// which is never more than arraySize.
size_t fillString(char *pArray, size_t arraySize);


vector<char> vc(maxNumChars);                         //
create a vector whose
                                                      // size
```

```
is maxNumChars

size_t charsWritten = fillString(&vc[0], vc.size());  // have
fillString write
                                                      // into
vc

string s(vc.begin(), vc.begin()+charsWritten);        // copy
data from vc to s
                                                      // via
range constructor
                                                      // (see
Item 5)
```

In fact, the idea of having a C API put data into a `vector` and then copying the data into the STL container you really want it in always works:

Click here to view code image

```
size_t fillArray(double *pArray, size_t arraySize);    // as
above

vector<double> vd(maxNumDoubles);                      //
also as above
vd.resize(fillArray(&vd[0], vd.size()));

deque<double> d(vd.begin(), vd.end());                 //
copy data into
                                                       //
deque

list<double> l(vd.begin(), vd.end());                  //
copy data into list

set<double> s(vd.begin(), vd.end());                   //
copy data into set
```

Furthermore, this hints at how STL containers other than `vector` or `string` can pass their data to C APIs. Just copy each container's data into a `vector`, then pass it to the API:

Click here to view code image

```
void doSomething(const int* pInts, size_t numInts);   // C
API (from above)

set<int> intSet;                                      // set
```

```
that will hold
...                                                     // data
to pass to API

vector<int> v(intSet.begin(), intSet.end());            // copy
set data into
                                                        // a
vector

if (!v.empty()) doSomething(&v[0], v.size());           // pass
the data to
                                                        // the
API
```

You could copy the data into an array, too, then pass the array to the C API, but why would you want to? Unless you know the size of the container during compilation, you'd have to allocate the array dynamically, and Item 13 explains why you should prefer `vector`s to dynamically allocated arrays anyway.

Item 17: Use "the `swap` trick" to trim excess capacity

So you're writing software to support the TV game show *Give Me Lots Of Money—Now!*, and you're keeping track of the potential contestants, whom you have stored in a vector:

```
class Contestant { ... };
vector<Contestant> contestants;
```

When the show puts out a call for new contestants, it's inundated with applicants, and your vector quickly acquires a lot of elements. As the show's producers vet the prospective players, however, a relatively small number of viable candidates get moved to the front of the vector (perhaps via `partial_sort` or `partition`—see Item 31), and the candidates no longer in the running are removed from the vector (typically by calling a range form of `erase`—see Item 5). This does a fine job of reducing the size of the vector, but it does nothing to reduce its capacity. If your vector held a hundred thousand potential candidates at some point, its capacity would continue to be at least 100,000, even if later it held only, say, 10.

To avoid having your vector hold onto memory it no longer needs, you'd like to have a way to reduce its capacity from the maximum it used to the amount it currently needs. Such a reduction in capacity is commonly known as “shrink to fit.” Shrink-to-fit is easy to program, but the code is—how shall I put this?—something less than intuitive. Let me show it to you, then I’ll explain how it works.

This is how you trim the excess capacity from your `contestants` vector:

```
vector<Contestant>(contestants.begin(),
contestants.end()).swap(contestants);
```

The expression `vector<Contestant>(contestants.begin(), contestants.end())` creates a temporary vector that is a copy of `contestants`; `vector`’s range constructor (see Item 5) does the work. The resulting temporary vector has no excess capacity. We then `swap` the data in the temporary vector with that in `contestants`. By the time we’re done, `contestants` has the trimmed capacity of the temporary, and the temporary holds the bloated capacity that used to be in `contestants`. At that point (the end of the statement), the temporary vector is destroyed, thus freeing the memory formerly used by `contestants`. *Voilà!* Shrink-to-fit.

The same trick is applicable to `string`s:

Click here to view code image

```
string s;

...                                     // make s large, then
erase most
                                        // of its characters

string(s.begin(), s.end()).swap(s);     // do a "shrink-to-
fit" on s
```

Now, the language police require that I inform you that there’s no guarantee that this technique will truly eliminate excess capacity. Implementers are free to give `vector`s and `string`s excess capacity if they want to, and sometimes they want to. For example, they may have a minimum capacity below which they never go, or they may constrain a `vector`’s or `string`’s capacity to be a power of two. (In my experience, such anomalies are more common in `string` implementations than in `vector` implementations. For examples, see Item 15.) This approach to “shrink-to-fit,” then, doesn’t really mean “make the capacity as small as possible,” it means “make the capacity as small as this implementation is willing to make it given the current size of the container.” Short of switching to a different implementation of the STL, however, this is the best you can do, so when you think “shrink-to-fit” for `vector`s and `string`s, think “the `swap` trick.”

As an aside, a variant of the `swap` trick can be used both to clear a container and to reduce its capacity to the minimum your implementation offers. You simply do the swap with a temporary `vector` or `string` that is default-constructed:

Click here to view code image

```
vector<Contestant> v;
string s;

...                                         // use v and s

vector<Contestant>().swap(v);               // clear v and
minimize its capacity
string().swap(s);                           // clear s and
minimize its capacity
```

Item 18: Avoid using `vector<bool>`

As an STL container, there are really only two things wrong with `vector<bool>`. First, it’s not an STL container. Second, it doesn’t hold `bool`s. Other than that, there’s not much to object to.

An object doesn’t become an STL container just because somebody says it’s one. An object becomes an STL container only if it fulfills all the container requirements laid down in section 23.1 of the Standard for C++. Among the requirements is that if `c` is a container of objects of type `T` and `c` supports `operator[]`, the following must compile:

Click here to view code image

```
T *p = &c[0];              // initialize a T* with the address
                           // of whatever operator[] returns
```

In other words, if you use `operator[]` to get at one of the `T` objects in a `Container<T>`, you can get a pointer to that object by taking its address. (This assumes that `T` hasn’t perversely overloaded `operator&`.) If `vector<bool>` is to be a container, then, this code must compile:

Click here to view code image

```
vector<bool> v;

bool *pb = &v[0];            // initialize a bool* with the
```

```
address of
                              // what vector<bool>::operator[]
returns
```

But it won’t compile. It won’t, because `vector<bool>` is a pseudo-container that contains not actual `bool`s, but a packed representation of `bool`s that is designed to save space. In a typical implementation, each “`bool`” stored in the “`vector`” occupies a single bit, and an eight-bit byte holds eight “`bool`s.” Internally, `vector<bool>` uses the moral equivalent of bitfields to represent the `bool`s it pretends to be holding.

Like `bool`s, bitfields represent only two possible values, but there is an important difference between true `bool`s and bitfields masquerading as `bool`s. You may create a pointer to a real `bool`, but pointers to individual bits are forbidden.

References to individual bits are forbidden, too, and this posed a problem for the design of `vector<bool>`’s interface, because the return type of `vector<T>::operator[]` is supposed to be `T&`. That wouldn’t be a problem if `vector<bool>` really held `bool`s, but because it doesn’t, `vector<bool>::operator[]` would need somehow to return a reference to a bit, and there’s no such thing.

To deal with this difficulty, `vector<bool>::operator[]` returns an object that *acts like* a reference to a bit, a so-called *proxy object*. (You don’t need to understand proxy objects to use the STL, but they’re a C++ technique worth knowing about. For information about them, consult Item 30 of *More Effective C++* as well as the “Proxy” chapter in Gamma *et al.*’s *Design Patterns* [6].) Stripped down to the bare essentials, `vector<bool>` looks like this:

Click here to view code image

```
template <typename Allocator>
vector<bool, Allocator> {
public:

  class reference { ... };              // class to generate
proxies for
                                        // references to
individual bits

  reference operator[](size_type n);    // operator[]
returns a proxy
```

```
  ...
};
```

Now it’s clear why this code won’t compile:

Click here to view code image

```
vector<bool> v;

bool *pb = &v[0];         // error! the expression on the
right is
                          // of type vector<bool>::reference*,
                          // not bool*
```

Because it won’t compile, `vector<bool>` fails to satisfy the requirements for STL containers. Yes, `vector<bool>` is in the Standard, and yes, it *almost* satisfies the requirements for STL containers, but almost just isn’t good enough. The more you write your own templates designed to work with the STL, the more you’ll appreciate that. The day will come, I assure you, when you’ll write a template that will work only if taking the address of a container element yields a pointer to the contained type, and when that day comes, you’ll suddenly understand the difference between being a container and *almost* being a container.

You may wonder why `vector<bool>` is in the Standard, given that it’s not a container and all. The answer has to do with a noble experiment that failed, but let me defer that discussion for a moment while I address a more pressing question. To wit, if `vector<bool>` should be avoided because it’s not a container, what should you use when you *need* a `vector<bool>`?

The standard library provides two alternatives that suffice almost all the time. The first is `deque<bool>`. A `deque` offers almost everything a `vector` does (the only notable omissions are `reserve` and `capacity`), and a `deque<bool>` is an STL container that really contains `bool`s. Of course, the underlying memory for a `deque` isn’t contiguous, so you can’t pass the data behind a `deque<bool>` to a C API[‡] that expects an array of `bool` (see Item 16), but you couldn’t do that with a `vector<bool>` anyway because there’s no portable way to get at the data for a `vector<bool>`. (The techniques that work for `vector`s in Item 16 fail to compile for `vector<bool>`, because they depend on being able to get a pointer to the type of element contained in the `vector`. Have I mentioned that a `vector<bool>` doesn’t contain `bool`s?)

[‡] This would presumably be a C99 API, because `bool` was added to C only as

of that version of the language.

The second alternative to `vector<bool>` is `bitset`. `bitset` isn't an STL container, but it is part of the standard C++ library. Unlike STL containers, its size (number of elements) is fixed during compilation, so there is no support for inserting or erasing elements. Furthermore, because it's not an STL container, it offers no support for iterators. Like `vector<bool>`, however, it uses a compact representation that devotes only a single bit to each value it contains. It offers `vector<bool>`'s special `flip` member function, as well as a number of other special member functions that make sense for collections of bits. If you can live without iterators and dynamic changes in size, you'll probably find that a `bitset` will serve you well.

And now let me discuss the noble experiment that failed, the one that left behind as its residue the STL non-container `vector<bool>`. I mentioned earlier that proxy objects are often useful in C++ software development. The members of the C++ Standardization Committee were well aware of this, so they decided to develop `vector<bool>` as a demonstration of how the STL could support containers whose elements are accessed via proxies. With this example in the Standard, they reasoned, practitioners would have a ready reference for implementing their own proxy-based containers.

Alas, what they discovered was that it was *not* possible to create proxy-based containers that satisfy all the requirements of STL containers. For one reason or another, however, their failed attempt at developing one remained in the Standard. One can speculate on why `vector<bool>` was retained, but practically speaking, it doesn't matter. What does matter is this: `vector<bool>` doesn't satisfy the requirements of an STL container; you're best off not using it; and `deque<bool>` and `bitset` are alternative data structures that will almost certainly satisfy your need for the capabilities promised by `vector<bool>`.

Chapter 3. Associative Containers

Somewhat like the polychromatic horse in *The Wizard of Oz* movie, the associative containers are creatures of a different color. True, they share many characteristics with the sequence containers, but they differ in a number of fundamental ways. For example, they automatically keep themselves sorted; they view their contents through the lens of equivalence instead of equality; `set`s and `map`s reject duplicate entries; and `map`s and `multimap`s generally ignore half of each object they contain. Yes, the associative containers are containers, but if you'll excuse my likening `vector`s and `string`s to Kansas, we are definitely not in Kansas any more.

In the Items that follow, I explain the critical notion of equivalence, describe an important restriction on comparison functions, motivate custom comparison functions for associative containers of pointers, discuss the official and practical sides of key `const`ness, and offer advice on improving the efficiency of associative containers.

At the end of the chapter, I examine the STL's lack of containers based on hash tables, and I survey two common (nonstandard) implementations. Though the STL proper doesn't offer hash tables, you need not write your own or do without. High-quality implementations are readily available.

Item 19: Understand the difference between equality and equivalence

The STL is awash in comparisons of objects to see if they have the same value. For example, when you ask `find` to locate the first object in a range with a particular value, `find` has to be able to compare two objects to see if the value of one is the same as the value of the other. Similarly, when you attempt to `insert` a new element into a `set`, `set::insert` has to be able to determine whether that element's value is already in the `set`.

The `find` algorithm and `set`'s `insert` member function are representative of many functions that must determine whether two values are the same. Yet they do it in different ways. `find`'s definition of "the same" is *equality*, which is based on `operator==`. `set::insert`'s definition of "the same" is *equivalence*, which is usually based on `operator<`. Because these are different definitions, it's possible for one definition to dictate that two objects have the same value while the other definition decrees that they do not. As a result, you must understand the difference between equality and equivalence if you are to make effective use of the STL.

Operationally, the notion of equality is based on `operator==`. If the expression "`x == y`" returns true, `x` and `y` have equal values, otherwise they don't. That's pretty straightforward, though it's useful to bear in mind that just because `x` and `y` have equal values does not necessarily imply that all of their data members have equal values. For example, we might have a `Widget` class that internally keeps track of its last time of access,

```
class Widget {
public:
  ...

private:
  TimeStamp lastAccessed;
  ...

};
```

and we might have an `operator==` for `Widget`s that ignores this field:

Click here to view code image

```
bool operator==(const Widget& lhs, const Widget& rhs)
{
  // code that ignores the lastAccessed field
}
```

In that case, two `Widget`s would have equal values even if their `lastAccessed` fields were different.

Equivalence is based on the relative ordering of object values in a sorted range. Equivalence makes the most sense if you think about it in terms of the sort order that is part of every standard associative container (i.e., `set`, `multiset`, `map`, and `multimap`). Two objects `x` and `y` have equivalent values with respect to the sort order used by an associative container `c` if neither precedes the other in `c`'s sort order. That sounds complicated, but in practice, it's not. Consider, as an example, a `set<Widget> s`. Two `Widget`s `w1` and `w2` have equivalent values with respect to `s` if neither precedes the other in `s`'s sort order. The default comparison function for a `set<Widget>` is `less<Widget>`, and by default `less<Widget>` simply calls `operator<` for `Widget`s, so `w1` and `w2` have equivalent values with respect to `operator<` if the following expression is true:

Click here to view code image

```
!(w1 < w2)                               // it's not true that
w1 < w2
&&                                       // and
!(w2 < w1)                               // it's not true that
w2 < w1
```

This makes sense: two values are equivalent (with respect to some ordering criterion) if neither precedes the other (according to that criterion).

In the general case, the comparison function for an associative container isn't `operator<` or even `less`, it's a user-defined predicate. (See Item 39 for more information on predicates.) Every standard associative container makes its sorting predicate available through its `key_comp` member function, so two objects `x` and `y` have equivalent values with respect to an associative container `c`'s sorting criterion if the following evaluates to `true`:

Click here to view code image

```
!c.key_comp()(x, y) && !c.key_comp()(y, x)  // it's not true
that x precedes
                                            // y in c's sort
order and it's
                                            // also not true
that y precedes
                                            // x in c's sort
order
```

The expression `!c.key_comp()(x, y)` looks nasty, but once you understand that `c.key_comp()` is returning a function (or a function object), the nastiness dissipates. `!c.key_comp()(x, y)` is just calling the function (or function object) returned by `key_comp`, and it's passing `x` and `y` as arguments. Then it negates the result. `c.key_comp()(x, y)` returns true only when `x` precedes `y` in `c`'s sort order, so `!c.key_comp()(x, y)` is true only when `x` *doesn't* precede `y` in `c`'s sort order.

To fully grasp the implications of equality versus equivalence, consider a case-insensitive `set<string>`, i.e., a `set<string>` where the set's comparison function ignores the case of the characters in the strings. Such a comparison function would consider "STL" and "stL" to be equivalent. Item 35 shows how to implement a function, `ciStringCompare`, that performs a case-insensitive comparison, but `set` wants a comparison function *type*, not an actual function. To bridge this gap, we write a functor class whose `operator()` calls `ciStringCompare`:

Click here to view code image

```
struct CIStringCompare:                            // class
for case-insensitive
  public                                           // string
comparisons;
  binary_function<string, string, bool> {          // see
Item 40 for info on
                                                   // this
base class
  bool operator()(const string& lhs,
                  const string& rhs) const
  {
    return ciStringCompare(lhs, rhs);              // see
Item 35 for how
  }                                                //
ciStringCompare is
                                                   //
implemented
};
```

Given `CIStringCompare`, it's easy to create a case-insensitive `set<string>`:

Click here to view code image

```
set<string, CIStringCompare> ciss;                 // ciss =
"case-insensitive
                                                   //
set"
```

If we insert `"Persephone"` and `"persephone"` into the set, only the first string is added, because the second one is equivalent to the first:

Click here to view code image

```
ciss.insert("Persephone");      // a new element is added to
the set

ciss.insert("persephone");      // no new element is added to
the set
```

If we now search for the string `"persephone"` using `set`'s `find` member function, the search will succeed,

```
if (ciss.find("persephone") != ciss.end()) ...      // this
```

```
test will succeed
```

but if we use the non-member `find` algorithm, the search will fail:

Click here to view code image

```
if (find(ciss.begin(), ciss.end(),
         "persephone") != ciss.end()) ...          // this
test will fail
```

That’s because `"persephone"` is *equivalent* to `"Persephone"` (with respect to the comparison functor `CIStringCompare`), but it’s not *equal* to it (because `string("persephone") != string("Persephone")`). This example demonstrates one reason why you should follow the advice in Item 44 and prefer member functions (like `set::find`) to their non-member counterparts (like `find`).

You might wonder why the standard associative containers are based on equivalence instead of equality. After all, most programmers have an intuition about equality that they lack for equivalence. (Were that not the case, there’d be no need for this Item.) The answer is simple at first glance, but the more closely you look, the more subtle it becomes.

The standard associative containers are kept in sorted order, so each container must have a comparison function (`less`, by default) that defines how to keep things sorted. Equivalence is defined in terms of this comparison function, so clients of a standard associative container need to specify only one comparison function (the one determining the sort order) for any container they use. If the associative containers used equality to determine when two objects have the same value, each associative container would need, in addition to its comparison function for sorting, a second comparison function for determining when two values are equal. (By default, this comparison function would presumably be `equal_to`, but it’s interesting to note that `equal_to` is never used as a default comparison function in the STL. When equality is needed in the STL, the convention is to simply call `operator==` directly. For example, this is what the non-member `find` algorithm does.)

Let’s suppose we had a `set`-like STL container called `set2CF`, “set with two comparison functions.” The first comparison function would be used to determine the sort order of the set, the second would be used to determine whether two objects had the same value. Now consider this `set2CF`:

```
set2CF<string, CIStringCompare, equal_to<string> > s;
```

Here, `s` sorts its strings internally without regard to case, and the equality criterion is the intuitive one: two strings have the same value if they are equal to one another. Let’s insert our two spellings of Hades’ reluctant bride (Persephone) into s:

```
s.insert("Persephone");
s.insert("persephone");
```

What should this do? If we observe that “`Persephone" != "persephone"` and insert them both into `s`, what order should they go in? Remember that the sorting function can’t tell them apart. Do we insert them in some arbitrary order, thus giving up the ability to traverse the contents of the set in deterministic order? (This inability to traverse associative container elements in a deterministic order already afflicts `multiset`s and `multimap`s, because the Standard places no constraints on the relative ordering of equivalent values (for `multiset`s) or keys (for `multimap`s).) Or do we insist on a deterministic ordering of `s`’s contents and ignore the second attempted insertion (the one for “persephone”)? If we do that, what happens here?

```
if (s.find("persephone") != s.end()) ...      // does this
test succeed or fail?
```

Presumably `find` employs the equality check, but if we ignored the second call to `insert` in order to maintain a deterministic ordering of the elements in `s`, this `find` will fail, even though the insertion of `"persephone"` was ignored on the basis of its being a duplicate value!

The long and short of it is that by using only a single comparison function and by employing equivalence as the arbiter of what it means for two values to be “the same,” the standard associative containers sidestep a whole host of difficulties that would arise if two comparison functions were allowed. Their behavior may seem a little odd at first (especially when one realizes that member and non-member `find` may return different results), but in the long run, it avoids the kind of confusion that would arise from mixing uses of equality and equivalence within standard associative containers.

Interestingly, once you leave the realm of *sorted* associative containers, the situation changes, and the issue of equality versus equivalence can be—and has been—revisited. There are two common designs for nonstandard (but widely available) associative containers based on hash tables. One design is based on equality, while the other is based on equivalence. I encourage you to turn to Item 25 to learn more about these containers and the design decisions they’ve adopted.

Item 20: Specify comparison types for associative containers of pointers

Suppose you have a `set` of `string*` pointers and you insert the names of some animals into the set:

Click here to view code image

```
set<string*> ssp;                                    // ssp =
"set of string ptrs"

ssp.insert(new string("Anteater"));
ssp.insert(new string("Wombat"));
ssp.insert(new string("Lemur"));
ssp.insert(new string("Penguin"));
```

You then write the following code to print the contents of the set, expecting the strings to come out in alphabetical order. After all, `set`s keep their contents sorted.

Click here to view code image

```
for (set<string*>::const_iterator i = ssp.begin();    // you
expect to see
     i != ssp.end();                                  //
this: "Anteater",
     ++i)                                             //
"Lemur", "Penguin",
  cout << *i << endl;                                 //
"Wombat"
```

The comment describes what you expect to see, but you don't see that at all. Instead, you see four hexadecimal numbers. They are pointer values. Because the set holds pointers, `*i` isn't a `string`, it's a *pointer* to a `string`. Let this be a lesson that reminds you to adhere to the guidance of Item 43 and avoid writing your own loops. If you'd used a call to the `copy` algorithm instead,

Click here to view code image

```
copy(ssp.begin(), ssp.end(),                          // copy
the strings in
     ostream_iterator<string>(cout, "\n"));           // ssp
to cout (but this
                                                      //
won't compile)
```

you'd not only have typed fewer characters, you'd have found out about your error sooner, because this call to `copy` won’t compile. `ostream_iterator` insists on knowing the type of object being printed, so when you tell it it’s a `string` (by passing that as the template parameter), your compilers detect the mismatch between that and the type of objects stored in `ssp` (which is `string*`), and they refuse to compile the code. Score another one for type safety.

If you exasperatedly change the `*i` in your explicit loop to `**i`, you *might* get the output you want, but you probably won’t. Yes, the animal names will be printed, but the chances of their coming out in alphabetical order are only 1 in 24. `ssp` keeps its contents in sorted order, but it holds pointers, so it sorts by *pointer value*, not by string value. There are 24 possible permutations for four pointer values, so there are 24 possible orders for the pointers to be stored in. Hence the odds of 1 in 24 of your seeing the strings in alphabetical order.[†]

[†] Practically speaking, the 24 permutations are not equally likely, so the “1 in 24” statement is a bit misleading. Still, there *are* 24 different orders, and you *could* get any one of them.

To surmount this problem, it helps to recall that

```
set<string*> ssp;
```

is shorthand for this:

```
set<string*, less<string*> > ssp;
```

Well, to be completely accurate, it’s shorthand for

```
set<string*, less<string*>, allocator<string*> > ssp;
```

but allocators don’t concern us in this Item, so we’ll ignore them.

If you want the `string*` pointers to be stored in the set in an order determined by the string values, you can’t use the default comparison functor class `less<string*>`. You must instead write your own comparison functor class, one whose objects take `string*` pointers and order them by the values of the `string`s they point to. Like this:

Click here to view code image

```
struct StringPtrLess:
  public binary_function<const string*,           // see
Item 40 for the
                         const string*,            // reason
for this base
                         bool> {                    // class
  bool operator()(const string *ps1, const string *ps2) const
  {
    return *ps1 < *ps2;
  }
};
```

Then you can use `StringPtrLess` as `ssp`'s comparison type:

[Click here to view code image](#)

```
typedef set<string*, StringPtrLess> StringPtrSet;

StringPtrSet ssp;                        // create a set of
strings and order
                                         // them as defined by
StringPtrLess

...                                      // insert the same
four strings as
                                         // before
```

Now your loop will finally do what you want it to do (provided you've fixed the earlier problem whereby you used `*i` instead of `**i`):

[Click here to view code image](#)

```
for (StringPtrSet::const_iterator i = ssp.begin();  // prints
"Anteater",
     i != ssp.end();                                //
"Lemur",
     ++i)                                           //
"Penguin",
   cout << **i << endl;                             //
"Wombat"
```

If you want to use an algorithm instead, you could write a function that knows how to dereference `string*` pointers before printing them, then use that function in conjunction with `for_each`:

[Click here to view code image](#)

```
void print(const string *ps)                        // print
to cout the
{                                                   // object
pointed to
cout << *ps << endl;                               // by ps
}

for_each(ssp.begin(), ssp.end(), print);            // invoke
```

```
print on each
                                                    //
element in ssp
```

Or you could get fancy and write a generic dereferencing functor class, then use that with `transform` and an `ostream_iterator`:

Click here to view code image

```
// when functors of this type are passed a T*, they return a
const T&
struct Dereference {

  template <typename T>
  const T& operator()(const T *ptr) const
  {
    return *ptr;
  }
};

transform(ssp.begin(), ssp.end(),                  //
"transform" each
          ostream_iterator<string>(cout, "\n"),  // element
in ssp by
          Dereference());                          //
dereferencing it,
                                                   // writing
the results
                                                   // to cout
```

Replacement of loops with algorithms, however, is not the point, at least not for this Item. (It *is* the point for Item 43.) The point is that anytime you create a standard associative container of pointers, you must bear in mind that the container will be sorted by the values of the pointers. Only rarely will this be what you want, so you'll almost always want to create your own functor class to serve as a comparison type.

Notice that I wrote "comparison *type*." You may wonder why you have to go to the trouble of creating a functor class instead of simply writing a comparison function for the set. For example, you might think to try this:

Click here to view code image

```
bool stringPtrLess(const string* ps1,     // would-be
comparison
                   const string* ps2)     // function for
string*
```

```
{                                         // pointers to be
sorted by
  return *ps1 < *ps2;                     // string value
}

set<string*, stringPtrLess> ssp;          // attempt to use
stringPtrLess
                                          // as ssp's
comparison function;
                                          // this won't
compile
```

The problem here is that each of the `set` template’s three parameters is a *type*. Unfortunately, `stringPtrLess` isn’t a type, it’s a function. That’s why the attempt to use `stringPtrLess` as `set`’s comparison function won’t compile. `set` doesn’t want a function, it wants a type that it can internally instantiate to *create* a function.

Anytime you create associative containers of pointers, figure you’re probably going to have to specify the container’s comparison type, too. Most of the time, your comparison type will just dereference the pointers and compare the pointed-to objects (as is done in `StringPtrLess` above). That being the case, you might as well keep a template for such comparison functors close at hand. Like this:

Click here to view code image

```
struct DereferenceLess {                   // see Item 7 for
why we're using
                                           // a class
containing a templatized
                                           // operator()
  template <typename PtrType>
  bool operator()(PtrType pT1,             // parameters are
passed by
                  PtrType pT2) const       // value, because
we expect them
  {                                        // to be (or to
act like) pointers
    return *pT1 < *pT2;
  }
};
```

Such a template eliminates the need to write classes like `StringPtrLess`, because we can use `DereferenceLess` instead:

Click here to view code image

```
set<string*, DereferenceLess> ssp;                // behaves
the same as
                                                  //
set<string*, StringPtrLess>
```

Oh, one more thing. This Item is about associative containers of pointers, but it applies equally well to containers of objects that *act* like pointers, e.g., smart pointers and iterators. If you have an associative container of smart pointers or of iterators, plan on specifying the comparison type for it, too. Fortunately, the solution for pointers tends to work for pointeresque objects, too. Just as `DereferenceLess` is likely to be suitable as the comparison type for an associative container of `T*`, it's likely to work as the comparison type for containers of iterators and smart pointers to `T` objects, too.

Item 21: Always have comparison functions return `false` for equal values

Let me show you something kind of cool. Create a `set` where `less_equal` is the comparison type, then insert 10 into the set:

```
set<int, less_equal<int> > s;        // s is sorted by "<="

s.insert(10);                        // insert the value 10
```

Now try inserting 10 again:

```
s.insert(10);
```

For this call to `insert`, the set has to figure out whether 10 is already present. We know that it is, but the set is dumb as toast, so it has to check. To make it easier to understand what happens when the set does this, we'll call the 10 that was initially inserted 10_A and the 10 that we're trying to insert 10_B.

The set runs through its internal data structures looking for the place to insert 10_B. It ultimately has to check 10_B to see if it's the same as 10_A. The definition of "the same" for associative containers is equivalence (see Item 19), so the set tests to see whether 10_B is equivalent to 10_A. When performing this test, it naturally uses the set's comparison function. In this example, that's `operator<=`, because we specified `less_equal` as the set's comparison function, and `less_equal` means `operator<=`. The set thus checks to see whether this expression is true:

```
!(10_A < = 10_B) && !(10_B <= 10_A)       // test 10_A and 10_B for
equivalence
```

Well, 10_A and 10_B are both 10, so it's clearly true that $10_A <= 10_B$. Equally clearly, $10_B <= 10_A$. The above expression thus simplifies to

```
!(true) && !(true)
```

and that simplifies to

```
false && false
```

which is simply `false`. That is, the set concludes that 10_A and 10_B are *not* equivalent, hence *not* the same, and it thus goes about inserting 10_B into the container alongside 10_A. Technically, this action yields undefined behavior, but the nearly universal outcome is that the set ends up with *two* copies of the value 10, and that means it's not a set any longer. By using `less_equal` as our comparison type, we've corrupted the container! Furthermore, *any* comparison function where equal values return `true` will do the same thing. Isn't that cool?

Okay, maybe your definition of cool isn't the same as mine. Even so, you'll still want to make sure that the comparison functions you use for associative containers always return `false` for equal values. You'll need to be vigilant, however. It's surprisingly easy to run afoul of this constraint.

For example, Item 20 describes how to write a comparison function for containers of `string*` pointers such that the container sorts its contents by the values of the `string`s instead of the values of the pointers. That comparison function sorts them in ascending order, but let's suppose you're in need of a comparison function for a container of `string*` pointers that sorts in descending order. The natural thing to do is to grab the existing code and modify it. If you're not careful, you might come up with this, where I've highlighted the changes to the code in Item 20:

Click here to view code image

```
struct StringPtrGreater:                            //
highlights show how
  public binary_function<const string*,             // this code
was changed
                         const string*,             // from page
89. Beware,
                         bool> {                    // this code
is flawed!

  bool operator()(const string *ps1, const string *ps2) const
  {
    return !( *ps1 < *ps2);                          // just
negate the old test;
  }                                                  // this is
incorrect!

};
```

The idea here is to reverse the sort order by negating the test inside the comparison function. Unfortunately, negating "<" doesn't give you ">" (which is what you want), it gives you ">=". And you now understand that ">=", because it will return `true` for equal values, is an invalid comparison function for associative containers.

The comparison type you really want is this one:

Click here to view code image

```
struct StringPtrGreater:                             // this is a
valid
  public binary_function<const string*,              //
comparison type for
                         const string*,              //
associative containers
                         bool> {

  bool operator()(const string *ps1, const string *ps2) const
  {
    return *ps2 < *ps1;                              // return
whether *ps2
  }                                                  // precedes
*ps1 (i.e., swap
                                                     // the order
of the
};                                                   // operands)
```

To avoid falling into this trap, all you need to remember is that the return value of a comparison function indicates whether one value precedes another in the sort order defined by that function. Equal values never precede one another, so comparison functions should always return `false` for equal values.

Sigh.

I know what you’re thinking. You’re thinking, “Sure, that makes sense for `set` and `map`, because those containers can’t hold duplicates. But what about `multiset` and `multimap`? Those containers may contain duplicates, so what do I care if the container thinks that two objects of equal value aren’t equivalent? It will store them both, which is what the `multi` containers are supposed to do. No problem, right?”

Wrong. To see why, let’s go back to the original example, but this time we’ll use a `multiset`:

Click here to view code image

```
multiset<int, less_equal<int> > s;       // s is still sorted
by "<="

s.insert(10);                            // insert 10A

s.insert(10);                            // insert 10B
```

`s` now has two copies of 10 in it, so we’d expect that if we do an `equal_range` on it, we’ll get back a pair of iterators that define a range containing both copies. But that’s not possible. `equal_range`, its name notwithstanding, doesn’t identify a range of equal values, it identifies a range of *equivalent* values. In this example, `s`’s comparison function says that 10_A and 10_B are not equivalent, so there’s no way that both can be in the range identified by `equal_range`.

You see? Unless your comparison functions always return `false` for equal values, you break all standard associative containers, regardless of whether they are allowed to store duplicates.

Technically speaking, comparison functions used to sort associative containers must define a “strict weak ordering” over the objects they compare. (Comparison functions passed to algorithms like `sort` (see Item 31) are similarly constrained.) If you’re interested in the details of what it means to be a strict weak ordering, you can find them in many comprehensive STL references, including Josuttis’ *The C++ Standard Library* [3], Austern’s *Generic Programming and the STL* [4], and the SGI STL Web Site [21]. I’ve never found the details terribly illuminating, but one of the requirements of a strict weak ordering bears directly on this Item. That requirement is that any function defining a strict weak ordering must return `false` if it’s passed two copies of the same value.

Hey! That *is* this Item!

Item 22: Avoid in-place key modification in `set` and `multiset`

The motivation for this Item is easy to understand. Like all the standard associative containers, `set` and `multiset` keep their elements in sorted order, and the proper behavior of these containers is dependent on their remaining sorted. If you change the value of an element in an associative container (e.g., change a 10 to a 1000), the new value might not be in the correct location, and that would break the sortedness of the container. Simple, right?

It's especially simple for `map` and `multimap`, because programs that attempt to change the value of a key in these containers won't compile:

Click here to view code image

```
map<int, string> m;
...
m.begin()->first = 10;           // error! map keys can't be
changed

multimap<int, string> mm;
...
mm.begin()->first = 20;          // error! multimap keys
can't
                                 // be changed, either
```

That's because the elements in an object of type `map<K, V>` or `multimap<K, V>` are of type `pair<const K, V>`. Because the type of the key is `const K`, it can't be changed. (Well, you can probably change it if you employ a `const_cast`, as we'll see below. Believe it or not, sometimes that's what you *want* to do.)

But notice that the title of this Item doesn't mention `map` or `multimap`. There's a reason for that. As the example above demonstrates, in-place key modification is impossible for `map` and `multimap` (unless you use a cast), but it may be possible for `set` and `multiset`. For objects of type `set<T>` or `multiset<T>`, the type of the elements stored in the container is simply `T`, not `const T`. Hence, the elements in a `set` or `multiset` may be changed anytime you want to. No cast is required. (Actually, things aren't quite that straightforward, but we'll come to that presently. There's no reason to get ahead of ourselves. First we crawl. Later we crawl on broken glass.)

Let us begin with an understanding of why the elements in a `set` or `multiset` aren’t `const`. Suppose we have a class for employees:

Click here to view code image

```
class Employee {
public:
  ...
  const string& name() const;           // get employee name
  void setName(const string& name);     // set employee name

  const string& title() const;          // get employee title
  void setTitle(const string& title);   // set employee title

  int idNumber() const;                 // get employee ID
number
  ...
};
```

As you can see, there are various bits of information we can get about employees. Let us make the reasonable assumption, however, that each employee has a unique ID number, a number that is returned by the `idNumber` function. To create a `set` of employees, then, it could easily make sense to order the set by ID number only, like this:

Click here to view code image

```
struct IDNumberLess:
  public binary_function<Employee, Employee, bool> {      //
see Item 40

  bool operator()(const Employee& lhs,
                  const Employee& rhs) const
  {
    return lhs.idNumber() < rhs.idNumber();
  }

};

typedef set<Employee, IDNumberLess> EmpIDSet;

EmpIDSet se;                           // se is a set of
employees
                                       // sorted by ID number
```

Practically speaking, the employee ID number is the *key* for the elements in this `set`. The rest of the employee data is just along for the ride. That being the case, there's no reason why we shouldn't be able to change the title of a particular employee to something interesting. Like so:

Click here to view code image

```
Employee selectedID;                           // variable to
hold employee
                                               // with
selected ID number
...

EmpIDSet::iterator i = se.find(selectedID);
if (i != se.end()){
    i->setTitle("Corporate Deity");            // give
employee new title;
}                                              // this
shouldn't compile --
                                               // see next
page for why
```

Because all we're doing here is changing an aspect of an employee that is unrelated to the way the set is sorted (a *non-key* part of the employee), this code can't corrupt the set. That's why it's reasonable. But such reasonable code precludes having the elements of a `set`/`multiset` be `const`. And that's why they aren't.

Why, you might wonder, doesn't the same logic apply to the keys in `map`s and `multimap`s? Wouldn't it make sense to create a map from employees to, say, the country in which they live, a map where the comparison function was `IDNumberLess`, just as in the previous example? And given such a map, wouldn't it be reasonable to change an employee's title without affecting the employee's ID number, just as in the previous example?

To be brutally frank, I think it would. Being equally frank, however, it doesn't matter what I think. What matters is what the Standardization Committee thought, and what it thought is that `map`/`multimap` keys should be `const` and `set`/`multiset` values shouldn't be.

Because the values in a `set` or `multiset` are not `const`, attempts to change them may compile. The purpose of this Item is to alert you that if you do change an element in a `set` or `multiset`, you must be sure not to change a *key part*—a part of the element that affects the sortedness of the container. If you do, you may corrupt the container, using the container will yield undefined results, and it will be your fault. On the other hand, this constraint applies only to the key parts of the contained objects. For all other parts of the contained elements, it’s open season; change away!

Except for the broken glass. Remember the broken glass I referred to earlier? We’re there now. Grab some bandages and follow me.

Even if `set` and `multiset` elements aren’t `const`, there are ways for implementations to keep them from being modified. For example, an implementation could have `operator*` for a `set<T>::iterator` return a `const T&`. That is, it could have the result of dereferencing a `set` iterator be a reference-to-*`const`* element of the set. Under such an implementation, there’d be no way to modify `set` or `multiset` elements, because all the ways of accessing the elements would add a `const` before letting you at them.

Are such implementations legal? Arguably yes. And arguably no. The Standard is inconsistent in this area, and in accord with Murphy’s Law, different implementers have interpreted it in different ways. The result is that it’s not uncommon to find STL implementations where this code compiles and others where it does not:

Click here to view code image

```
EmpIDSet se;                            // as before, se is a
set of employees
                                        // sorted by ID
number

Employee selectedID;                    // as before,
selectedID is a dummy
                                        // employee with the
selected ID
                                        // number
...

EmpIDSet::iterator i = se.find(selectedID);
if (i != se.end()){
    i->setTitle("Corporate Deity");     // some STL
implementations
}                                       // accept this line,
others reject it
```

Because of the equivocal state of the Standard and the differing interpretations it has engendered, code that attempts to modify elements in a `set` or `multiset` isn’t portable.[†]

[‡] The Standardization Committee has since clarified that elements in a `set` or `map` should not be modifiable without a cast. However, versions of the STL implemented prior to this clarification continue to be used, so, practically speaking, the material in this Item continues to apply.

So where do we stand? Encouragingly, things aren’t complicated:

- If portability is not a concern, you want to change the value of an element in a `set` or `multiset`, and your STL implementation will let you get away with it, go ahead and do it. Just be sure not to change a key part of the element, i.e., a part of the element that could affect the sortedness of the container.
- If you value portability, assume that the elements in `set`s and `multiset`s cannot be modified, at least not without a cast.

Ah, casts. We’ve seen that it can be entirely reasonable to change a non-key portion of an element in a `set` or a `multiset`, so I feel compelled to show you how to do it. How to do it correctly and portably, that is. It’s not hard, but it requires a subtlety too many programmers overlook: you must cast to a *reference*. As an example, look again at the `setTitle` call we just saw that failed to compile under some implementations:

Click here to view code image

```
EmpIDSet::iterator i = se.find(selectedID);
if (i != se.end()) {
    i->setTitle("Corporate Deity");    // some STL
implementations will
}                                      // reject this line
because *i is const
```

To get this to compile and behave correctly, we must cast away the constness of `*i`. Here’s the correct way to do it:

Click here to view code image

```
if (i != se.end())
{                                           // cast away
    const_cast<Employee&>(*i).setTitle("Corporate
Deity");     // constness
}
of *i
```

This takes the object pointed to by `i`, tells your compilers to treat the result of the cast as a reference to a (non-`const`) `Employee`, then invoke `setTitle` on the reference. Rather than explain why this works, I'll explain why an alternative approach fails to behave the way people often expect it to.

Many people come up with this code,

[Click here to view code image]

```
if (i != se.end())
{                                                // cast *i
    static_cast<Employee>(*i).setTitle("Corporate
Deity");       // to an
}
Employee
```

which is equivalent to the following:

[Click here to view code image]

```
if (i != se.end())
{                                            // same as above,
   ((Employee)(*i)).setTitle("Corporate
Deity");             // but using C
}
cast syntax
```

Both of these compile, and because these are equivalent, they're wrong for the same reason. At runtime, they don't modify `*i`! In both cases, the result of the cast is a temporary anonymous object that is a *copy* of `*i`, and `setTitle` is invoked on the anonymous object, not on `*i`! `*i` isn't modified, because `setTitle` is never invoked on that object, it's invoked on a *copy* of that object. Both syntactic forms are equivalent to this:

[Click here to view code image]

```
if (i != se.end()) {
    Employee tempCopy(*i);                      // copy *i into
tempCopy
    tempCopy.setTitle("Corporate Deity");      // modify
tempCopy
}
```

Now the importance of a cast to a reference should be clear. By casting to a reference, we avoid creating a new object. Instead, the result of the cast is a reference to an *existing* object, the object pointed to by `i`. When we invoke `setTitle` on the object indicated by the reference, we're invoking `setTitle` on `*i`, and that's exactly what we want.

What I've just written is fine for `set`s and `multiset`s, but when we turn to `map`s and `multimap`s, the plot thickens. Recall that a `map<K, V>` or a `multimap<K, V>` contains elements of type `pair<const K, V>`. That `const` means that the first component of the pair is *defined* to be `const`, and *that* means that attempts to modify it (even after casting away its constness) are undefined. If you're a stickler for following the rules laid down by the Standard, you'll *never* try to modify an object serving as a key to a `map` or `multimap`.

You've surely heard that casts are dangerous, and I hope this book makes clear that I believe you should avoid them whenever you can. To perform a cast is to shuck temporarily the safety of the type system, and the pitfalls we've discussed exemplify what can happen when you leave your safety net behind.

Most casts can be avoided, and that includes the ones we've just considered. If you want to change an element in a `set`, `multiset`, `map`, or `multimap` in a way that always works and is always safe, do it in five simple steps:

1. Locate the container element you want to change. If you're not sure of the best way to do that, Item 45 offers guidance on how to perform an appropriate search.
2. Make a copy of the element to be modified. In the case of a `map` or `multimap`, be sure not to declare the first component of the copy `const`. After all, you want to change it!
3. Modify the copy so it has the value you want to be in the container.
4. Remove the element from the container, typically via a call to `erase` (see Item 9).
5. Insert the new value into the container. If the location of the new element in the container's sort order is likely to be the same or adjacent to that of the removed element, use the "hint" form of `insert` to improve the efficiency of the insertion from logarithmic-time to amortized constant-time. Use the iterator you got from Step 1 as the hint.

Here's the same tired employee example, this time written in a safe, portable manner:

Click here to view code image

```
EmpIDSet se;                        // as before, se is a
set of employees
                                    // sorted by ID number
```

```
Employee selectedID;                      // as before,
selectedID is a dummy
                                          // employee with the
desired ID number
...

EmpIDSet::iterator i =
se.find(selectedID);                     // Step 1: find element
to change

if (i != se.end()) {
    Employee e(*i);                       // Step 2: copy the
element

    e.setTitle("Corporate Deity");        // Step 3: modify the
copy

    se.erase(i++);                        // Step 4: remove the
element;
                                          // increment the
iterator to maintain
                                          // its validity (see
Item 9)

    se.insert(i, e);                      // Step 5: insert new
value; hint that its
                                          // location is the same
as that of the
}                                         // original element
```

You’ll excuse my putting it this way, but the *key* thing to remember is that with `set` and `multiset`, if you perform any in-place modifications of container elements, you are responsible for making sure that the container remains sorted.

Item 23: Consider replacing associative containers with sorted `vector`s

Many STL programmers, when faced with the need for a data structure offering fast lookups, immediately think of the standard associative containers, `set`, `multiset`, `map`, and `multimap`. That’s fine, as far as it goes, but it doesn’t go far enough. If lookup speed is really important, it’s almost certainly worthwhile to consider the nonstandard hashed containers as well (see Item 25). With suitable hashing functions, hashed containers can be expected to offer constant-time lookups. (With poorly chosen hashing functions or with table sizes that are too small, the performance of hash table lookups may degrade significantly, but this is relatively uncommon in practice.) For many applications, the expected constant-time lookups of hashed containers are preferable to the guaranteed logarithmic-time lookups that are the hallmark of `set`, `map` and their `multi` companions.

Even if logarithmic-time lookup is what you want, the standard associative containers may not be your best bet. Counterintuitively, it's not uncommon for the standard associative containers to offer performance that is inferior to that of the lowly `vector`. If you want to make effective use of the STL, you need to understand when and how a `vector` can offer faster lookups than a standard associative container.

The standard associative containers are typically implemented as balanced binary search trees. A balanced binary search tree is a data structure that is optimized for a mixed combination of insertions, erasures, and lookups. That is, it's designed for applications that do some insertions, then some lookups, then maybe some more insertions, then perhaps some erasures, then a few more lookups, then more insertions or erasures, then more lookups, etc. The key characteristic of this sequence of events is that the insertions, erasures, and lookups are all mixed up. In general, there's no way to predict what the next operation on the tree will be.

Many applications use their data structures in a less chaotic manner. Their use of data structures fall into three distinct phases, which can be summarized like this:

1. **Setup.** Create a new data structure by inserting lots of elements into it. During this phase, almost all operations are insertions and erasures. Lookups are rare or nonexistent.
2. **Lookup.** Consult the data structure to find specific pieces of information. During this phase, almost all operations are lookups. Insertions and erasures are rare or nonexistent. There are so many lookups, the performance of this phase makes the performance of the other phases incidental.
3. **Reorganize.** Modify the contents of the data structure, perhaps by erasing all the current data and inserting new data in its place.

> Behaviorally, this phase is equivalent to phase 1. Once this phase is completed, the application returns to phase 2.

For applications that use their data structures in this way, a `vector` is likely to offer better performance (in both time and space) than an associative container. But not just any `vector` will do. It has to be a *sorted* `vector`, because only sorted containers work correctly with the lookup algorithms `binary_search`, `lower_bound`, `equal_range`, etc. (see Item 34). But why should a binary search through a (sorted) vector offer better performance than a binary search through a binary search tree? Because some things are trite but true, and one of them is that size matters. Others are less trite but no less true, and one of those is that locality of reference matters, too.

Consider first the size issue. Suppose we need a container to hold `Widget` objects, and, because lookup speed is important to us, we are considering both an associative container of `Widget`s and a sorted `vector<Widget>`. If we choose an associative container, we'll almost certainly be using a balanced binary tree. Such a tree would be made up of tree nodes, each holding not only a `Widget`, but also a pointer to the node's left child, a pointer to its right child, and (typically) a pointer to its parent. That means that the space overhead for storing a `Widget` in an associative container would be at least three pointers.

In contrast, there is no overhead when we store a `Widget` in a `vector`; we simply store a `Widget`. The `vector` itself has overhead, of course, and there may be empty (reserved) space at the end of the `vector` (see Item 14), but the per-`vector` overhead is typically insignificant (usually three machine words, e.g., three pointers or two pointers and an `int`), and the empty space at the end can be lopped off via "the `swap` trick" if necessary (see Item 17). Even if the extra space is not eliminated, it's unimportant for the analysis below, because that memory won't be referenced when doing a lookup.

Assuming our data structures are big enough, they'll be split across multiple memory pages, but the `vector` will require fewer pages than the associative container. That's because the `vector` requires no per-`Widget` overhead, while the associative container exacts three pointers per `Widget`. To see why this is important, suppose you're working on a system where a `Widget` is 12 bytes in size, pointers are 4 bytes, and a memory page holds 4096 (4K) bytes. Ignoring the per-container overhead, you can fit 341 `Widget`s on a page when they are stored in a `vector`, but you can fit at most 170 when they are stored in an associative container. You'll thus use about twice as much memory for the associative container as you would for the `vector`. If you're working in an environment where virtual memory is available, it's easy to see how that can translate into a lot more page faults, therefore a system that is significantly slower for large sets of data.

I'm actually being optimistic about the associative containers here, because I'm assuming that the nodes in the binary trees are clustered together on a relatively small set of memory pages. Most STL implementations use custom memory managers (implemented on top of the containers' allocators—see Items 10 and 11) to achieve such clustering, but if your STL implementation fails to take steps to improve locality of reference among tree nodes, the nodes could end up scattered all over your address space. That would lead to even more page faults. Even with the customary clustering memory managers, associative containers tend to have more problems with page faults, because, unlike contiguous-memory containers such as `vector`, node-based containers find it more difficult to guarantee that container elements that are close to one another in a container's traversal order are also close to one another in physical memory. Yet this is precisely the kind of memory organization that minimizes page faults when performing a binary search.

Bottom line: storing data in a sorted `vector` is likely to consume less memory than storing the same data in a standard associative container, and searching a sorted `vector` via binary search is likely to be faster than searching a standard associative container when page faults are taken into account.

Of course, the big drawback of a sorted `vector` is that it must remain sorted! When a new element is inserted, everything beyond the new element must be moved up by one. That's as expensive as it sounds, and it gets even more expensive if the vector has to reallocate its underlying memory (see Item 14), because then *all* the elements in the vector typically have to be copied. Similarly, if an element is removed from the vector, all the elements beyond it must be moved down. Insertions and erasures are expensive for `vector`s, but they're cheap for associative containers. That's why it makes sense to consider using a sorted `vector` instead of an associative container only when you know that your data structure is used in such a way that lookups are almost never mixed with insertions and erasures.

This Item has featured a lot of text, but it's been woefully short on examples, so let’s take a look at a code skeleton for using a sorted `vector` instead of a `set`:

Click here to view code image

```
vector<Widget> vw;                               //
alternative to set<Widget>

...                                              // Setup
phase: lots of insertions,
                                                 // few
lookups

sort(vw.begin(), vw.end());                      // end of
Setup phase. (When
                                                 //
simulating a multiset, you might
                                                 // prefer
stable_sort instead see
                                                 // Item 31.)
When emulating a set,
                                                 // be sure
vw holds no duplicates!

Widget w;                                        // object
for value to look up

...                                              // start
Lookup phase

if (binary_search(vw.begin(), vw.end(), w)) ...  // lookup
via binary_search

vector<Widget>::iterator i =
  lower_bound(vw.begin(), vw.end(), w);          // lookup
via lower_bound;
if (i != vw.end() && !(w < *i)) ...              // see Item
19 for an explana-
                                                 // tion of
the "!(w < *i)" test

pair<vector<Widget>::iterator,
     vector<Widget>::iterator> range =
  equal_range(vw.begin(), vw.end(), w);          // lookup
via equal_range
if (range.first != range.second) ...
```

```
...                                              // end
Lookup phase, start
                                                 //
Reorganize phase

sort(vw.begin(), vw.end());                      // begin new
Lookup phase...
```

As you can see, it's all pretty straightforward. The hardest thing about it is deciding among the search algorithms (e.g., `binary_search`, `lower_bound`, etc.), and Item 45 helps you do that.

Things get a bit more interesting when you decide to replace a `map` or `multimap` with a `vector`, because the `vector` must hold `pair` objects. After all, that's what `map` and `multimap` hold. Recall, however, that if you declare an object of type `map<K, V>` (or its `multimap` equivalent), the type of elements stored in the map is `pair<const K, V>`. To emulate a `map` or `multimap` using a `vector`, you must omit the `const`, because when you sort the vector, the values of its elements will get moved around via assignment, and that means that both components of the pair must be assignable. When using a `vector` to emulate a `map<K, V>`, then, the type of the data stored in the vector will be `pair<K, V>`, not `pair<const K, V>`.

`map`s and `multimap`s keep their elements in sorted order, but they look only at the key part of the element (the first component of the pair) for sorting purposes, and you must do the same when sorting a `vector`. You'll need to write a custom comparison function for your pairs, because `pair`'s `operator<` looks at *both* components of the `pair`.

Interestingly, you'll need a second comparison function for performing lookups. The comparison function you'll use for sorting will take two `pair` objects, but lookups are performed given only a key value. The comparison function for lookups, then, must take an object of the key type (the value being searched for) and a `pair` (one of the pairs stored in the vector)—two different types. As an additional twist, you can't know whether the key value or the `pair` will be passed as the first argument, so you really need two comparison functions for lookups: one where the key value is passed first and one where the `pair` is passed first.

Here's an example of how to put all the pieces together:

Click here to view code image

```
typedef pair<string, int> Data;                         //
type held in the "map"
                                                        // in
this example

class DataCompare {                                     //
class for comparison
public:                                                 //
functions

  bool operator()(const Data&
lhs,                            // comparison func
                  const Data& rhs)
const                      // for sorting
  {
    return keyLess(lhs.first,
rhs.first);                     // keyLess is below
  }

  bool operator()(const Data&
lhs,                            // comparison func
                  const Data::first_type& k)
const            // for lookups
  {
(form 1)
    return keyLess(lhs.first, k);
  }

  bool operator()(const Data::first_type&
k,                  // comparison func
                  const Data& rhs)
const                      // for lookups
  {
(form 2)
    return keyLess(k, rhs.first);
  }

private:
  bool keyLess(const Data::first_type&
k1,                    // the "real"
               const Data::first_type& k2)
const              // comparison
  {
function
    return k1 < k2;
  }
};
```

In this example, we assume that our sorted vector will be emulating a `map<string, int>`. The code is pretty much a literal translation of the discussion above, except for the presence of the member function `keyLess`. That function exists to ensure consistency among the various `operator()` functions. Each such function simply compares two key values, so rather than duplicate the logic, we put the test inside `keyLess` and have the `operator()` functions return whatever `keyLess` does. This admirable act of software engineering enhances the maintainability of `DataCompare`, but there is a minor drawback. Its provision for `operator()` functions with different parameter types renders the resulting function objects unadaptable (see Item 40). Oh well.

Using a sorted `vector` as a map is essentially the same as using it as a set. The only major difference is the need to use `DataCompare` objects as comparison functions:

Click here to view code image

```
vector<Data> vd;                                //
alternative to
                                                //
map<string, int>

...                                             // Setup
phase: lots of
                                                //
insertions, few lookups

sort(vd.begin(), vd.end(), DataCompare());      // end of
Setup phase. Simu-
                                                // lated maps
must avoid
                                                // dupes;
simulated multimaps
                                                // might use
stable_sort

string s;                                       // object for
value to look up

...                                             // start
Lookup phase
```

```
if (binary_search(vd.begin(), vd.end(), s,
                  DataCompare())) ...           // lookup via
binary_search

vector<Data>::iterator i =
  lower_bound(vd.begin(), vd.end(), s,
              DataCompare());                   // lookup via
lower_bound;
if (i != vd.end() && !DataCompare()(s, *i))...  // again, see
Item 19 for info
                                                // on the
                                                //
"!DataCompare()(s, *i)" test

pair<vector<Data>::iterator,
     vector<Data>::iterator> range =
  equal_range(vd.begin(), vd.end(), s,
              DataCompare());                   // lookup via
equal_range
if (range.first != range.second) ...

...                                             // end Lookup
phase, start
                                                // Reorganize
phase

sort(vd.begin(), vd.end(), DataCompare());      // begin new
Lookup phase...
```

As you can see, once you've written `DataCompare`, things pretty much fall into place. And once in place, they'll often run faster and use less memory than the corresponding design using a real `map` *as long as your program uses the data structure in the phased manner described on page 101*. If your program doesn't operate on the data structure in a phased manner, use of a sorted `vector` instead of a standard associative container will almost certainly end up wasting time.

Item 24: Choose carefully between `map::operator[]` and `map::insert` when efficiency is important

Let's suppose we have a `Widget` class that supports default construction as well as construction and assignment from a `double`:

[Click here to view code image]

```
class Widget {
public:
  Widget();
  Widget(double weight);

  Widget& operator=(double weight);
  ...
};
```

Let's now suppose we'd like to create a `map` from `int`s to `Widget`s, and we'd like to initialize the map with particular values. 'Tis simplicity itself:

```
map<int, Widget> m;

m[1] = 1.50;
m[2] = 3.67;
m[3] = 10.5;
m[4] = 45.8;
m[5] = 0.0003;
```

In fact, the only thing simpler is forgetting what's really going on. That's too bad, because what's going on could incur a considerable performance hit.

The `operator[]` function for `map`s is a novel creature, unrelated to the `operator[]` functions for `vector`s, `deque`s, and `string`s and equally unrelated to the built-in `operator[]` that works with arrays. Instead, `map::operator[]` is designed to facilitate "add or update" functionality. That is, given

```
map<K, V> m;
```

the expression

```
m[k] = v;
```

checks to see if the key `k` is already in the map. If not, it's added, along with `v` as its corresponding value. If `k` is already in the map, its associated value is updated to `v`.

The way this works is that `operator[]` returns a reference to the value object associated with `k`. `v` is then assigned to the object to which the reference (the one returned from `operator[]`) refers. This is straightforward when an existing key's associated value is being updated, because there's already a value object to which `operator[]` can return a reference. But if `k` isn't yet in the map, there's no value object for `operator[]` to refer to. In that case, it creates one from scratch by using the value type's default constructor. `operator[]` then returns a reference to this newly-created object.

Let’s look again at the first part of our original example:

```
map<int, Widget> m;

m[1] = 1.50;
```

The expression `m[1]` is shorthand for `m.operator[](1)`, so this is a call to `map::operator[]`. That function must return a reference to a `Widget`, because `m`’s mapped type is `Widget`. In this case, `m` doesn’t yet have anything in it, so there is no entry in the map for the key `1`. `operator[]` therefore default-constructs a `Widget` to act as the value associated with `1`, then returns a reference to that Widget. Finally, the Widget becomes the target of an assignment; the assigned value is `1.50`.

In other words, the statement

```
m[1] = 1.50;
```

is functionally equivalent to this:

Click here to view code image

```
typedef map<int, Widget> IntWidgetMap;              //
convenience
                                                    //
typedef

pair<IntWidgetMap::iterator, bool> result =         // create
new map
  m.insert(IntWidgetMap::value_type(1, Widget()));  // entry
with key 1
                                                    // and a
default-
                                                    //
constructed value
                                                    //
object; see below
                                                    // for a
comment on
                                                    //
value_type

result.first->second = 1.50;                        // assign
to the
                                                    // newly-
```

```
constructed
                                                                 // value
object
```

Now it should be clear why this approach may degrade performance. We first default-construct a `Widget`, then we immediately assign it a new value. If it's measurably more efficient to construct a `Widget` with the value we want instead of default-constructing the `Widget` and then doing the assignment, we'd be better off replacing our use of `operator[]` (including its attendant construction plus assignment) with a straightforward call to `insert`:

```
m.insert(IntWidgetMap::value_type(1, 1.50));
```

This has precisely the same ultimate effect as the code above, except it replaces construction (and later destruction) of a temporary *plus an assignment* with construction of a temporary from a pair of values (followed by later destruction of that temporary); there's no assignment. The more expensive the cost of assignment of the value part of the pair, the more you save by using `map::insert` instead of `map::operator[]`.

The code above takes advantage of the `value_type` typedef that's provided by every standard container. There's nothing particularly significant about this typedef, but it's important to remember that for `map` and `multimap` (as well as the nonstandard containers `hash_map` and `hash_multimap`—see Item 25), the type of the contained elements will always be some kind of `pair`.

I remarked earlier that `operator[]` is designed to facilitate "add or update" functionality, and we now understand that when an "add" is performed, `insert` is more efficient than `operator[]`. The situation is reversed when we do an update, i.e., when an equivalent key (see Item 19) is already in the map. To see why that is the case, look at our update options:

Click here to view code image

```
m[k] =
v;                                                                // use
operator[]
                                                                         /
to update k's
                                                                         /
value to be v

m.insert(IntWidgetMap::value_type(k, v)).first->second = v;
// use insert to
```

```
                                                                       /
update k's
                                                                       /
value to be v
```

The syntax alone is probably enough to convince you to favor `operator[]`, but we’re focusing on efficiency here, so we’ll overlook that.

The call to `insert` requires an argument of type `IntWidgetMap::value_type` (i.e., `pair<int, Widget>`), so when we call `insert`, we must construct and destruct an object of that type. That costs us a `pair` constructor and destructor. That, in turn, entails a `Widget` construction and destruction, because a `pair<int, Widget>` itself contains a `Widget` object. `operator[]` uses no `pair` object, so it constructs and destructs no pair and no `Widget`.

Efficiency considerations thus lead us to conclude that `insert` is preferable to `operator[]` when adding an element to a map, and both efficiency and aesthetics dictate that `operator[]` is preferable when updating the value of an element that’s already in the map.

It would be nice if the STL provided a function that offered the best of both worlds, i.e., efficient add-or-update functionality in a syntactically attractive package. For example, it’s not hard to envision a calling interface such as this:

Click here to view code image

```
iterator affectedPair =                 // if key k isn't in
map m, efficiently
  efficientAddOrUpdate(m, k, v);        // add pair (k,v) to
m; otherwise
                                        // efficiently update
to v the value
                                        // associated with k.
Return an
                                        // iterator to the
added or
                                        // modified pair
```

There’s no function like this in the STL, but, as the code below demonstrates, it’s not terribly hard to write yourself. The comments summarize what’s going on, and the paragraphs that follow provide some additional explanation.

Click here to view code image

```
template< typename MapType,                // type of map
         typename KeyArgType,             // see below for
why
         typename ValueArgType>           // KeyArgType and
                                          // ValueArgType are
type
typename MapType::iterator                // parameters
efficientAddOrUpdate(MapType& m,
                     const KeyArgType& k,
                     const ValueArgType& v)
{
  typename MapType::iterator lb =         // find where k is
or should
  m.lower_bound(k);                       // be; see page 7B
for why
                                          // "typename" is
needed
                                          // here

  if (lb != m.end() &&                    // if lb points to
a pair
      !(m.key_comp()(k, lb->first))) {    // whose key is
equiv to k...

     lb->second = v;                      // update the
pair's value
     return lb;                           // and return an
iterator
  }                                       // to that pair

  else {
    typedef typename MapType::value_type MVT;
    return m.insert(lb, MVT(k, v));       // add pair(k, v)
to m and
  }                                       // return an
iterator to the
}                                         // new map element
```

To perform an efficient add or update, we need to be able to find out if `k`'s value is in the map; if so, where it is; and if not, where it should be inserted. That job is tailor-made for `lower_bound` (see Item 45), so that's the function we call. To determine whether `lower_bound` found an element with the key we were looking for, we perform the second half of an equivalence test (see Item 19), being sure to use the correct comparison function for the map; that comparison function is available via `map::key_comp`. The result of the equivalence test tells us whether we're performing an add or an update.

If it's an update, the code is straightforward. The insert branch is more interesting, because it uses the "hint" form of `insert`. The construct `m.insert(lb, MVT(k, v))` "hints" that `lb` identifies the correct insertion location for a new element with key equivalent to `k`, and the Standard guarantees that if the hint is correct, the `insert`ion will occur in amortized constant, rather than logarithmic, time. In `efficientAddOrUpdate`, we know that `lb` identifies the proper insertion location, so the call to `insert` is guaranteed to be an amortized constant-time operation.

An interesting aspect of this implementation is that `KeyArgType` and `ValueArgType` need not be the types stored in the map. They need only be *convertible* to the types stored in the map. The alternative would be to eliminate the type parameters `KeyArgType` and `ValueArgType`, using instead `MapType::key_type` and `MapType::mapped_type`. However, if we did that, we might force unnecessary type conversions to occur at the point of call. For instance, look again at the definition of the map we've been using in this Item's examples:

```
map<int, Widget> m;                          // as before
```

And recall that `Widget` accepts assignment from a `double`:

Click here to view code image

```
class Widget {                               // also as
before
public:
  ...
  Widget& operator=(double weight);
  ...
};
```

Now consider this call to `efficientAddOrUpdate`:

```
efficientAddOrUpdate(m, 10, 1.5);
```

Suppose that it's an update operation, i.e., `m` already contains an element whose key is 10. In that case, the template above deduces that `ValueArgType` is `double`, and the body of the function directly assigns 1.5 as a `double` to the `Widget` associated with the key 10. That's accomplished by an invocation of `Widget::operator=(double)`. If we had used `MapType::mapped_type` as the type of `efficientAddOrUpdate`'s third parameter, we'd have converted 1.5 to a `Widget` at the point of call, and we'd thus have paid for a `Widget` construction (and subsequent destruction) we didn't need.

Subtleties in the implementation of `efficientAddOrUpdate` may be interesting, but they're not as important as the main point of this Item, which is that you should choose carefully between `map::operator[]` and `map::insert` when efficiency is important. If you're updating an existing `map` element, `operator[]` is preferable, but if you're adding a new element, `insert` has the edge.

Item 25: Familiarize yourself with the nonstandard hashed containers

It generally doesn't take much time for STL programmers to begin to wonder, "Vectors, lists, maps, sure, but where are the hash tables?" Alas, there aren't any hash tables in the standard C++ library. Everyone agrees that this is unfortunate, but the Standardization Committee felt that the work needed to add them would have unduly delayed completion of the Standard. It's a foregone conclusion that the next version of the Standard will include hash tables, but for the time being, the STL doesn't do hashing.

If you like hash tables, however, take heart. You need not do without or roll your own. STL-compatible hashed associative containers are available from multiple sources, and they even have *de facto* standard names: `hash_set`, `hash_multiset`, `hash_map`, and `hash_multimap`.

Behind these common names, different implementations, er, differ. They differ in their interfaces, their capabilities, their underlying data structures, and the relative efficiency of the operations they support. It's still possible to write reasonably portable code using hash tables, but it's not as easy as it would be had the hashed containers been standardized. (Now you know why standards are important.)

Of the several available implementations for hashed containers, the two most common are from SGI (see Item 50) and Dinkumware (see Appendix B), so in what follows, I restrict myself to the designs of the hashed containers from these vendors. STLport (again, see Item 50) also offers hashed containers, but the STLport hashed containers are based on those from SGI. For purposes of this Item, assume that whatever I write about the SGI hashed containers applies to the STLport hashed containers, too.

Hashed containers are associative containers, so it should not surprise you that, like all associative containers, they need to know the type of objects stored in the container, the comparison function for such objects, and the allocator for these objects. In addition, hashed containers require specification of a hashing function. That suggests the following declaration for hashed containers:

Click here to view code image

```
template<typename T,
         typename HashFunction,
         typename CompareFunction,
         typename Allocator = allocator<T> >
class hash_container;
```

This is quite close to the SGI declaration for hashed containers, the primary difference being that SGI provides default types for `HashFunction` and `CompareFunction`. The SGI declaration for `hash_set` looks essentially like this (I've tweaked it a bit for presentation purposes):

Click here to view code image

```
template<typename T,
         typename HashFunction = hash<T>,
         typename CompareFunction = equal_to<T>,
         typename Allocator = allocator<T> >
class hash_set;
```

A noteworthy aspect of the SGI design is the use of `equal_to` as the default comparison function. This is a departure from the conventions of the standard associative containers, where the default comparison function is `less`. This design decision signifies more than a simple change in default comparison functions. SGI's hashed containers determine whether two objects in a hashed container have the same value by testing for *equality*, not equivalence (see Item 19). For hashed containers, this is not an unreasonable decision, because hashed associative containers, unlike their standard (typically tree-based) counterparts, are not kept in sorted order.

The Dinkumware design for hashed containers adopts some different strategies. It still allows you to specify object types, hash function types, comparison function types, and allocator types, but it moves the default hash and comparison functions into a separate traits-like class called `hash_compare`, and it makes `hash_compare` the default argument for the `HashingInfo` parameter of the container templates. (If you're unfamiliar with the notion of a "traits" class, open a good STL reference like Josuttis' *The C++ Standard Library*

[3] and study the motivation and implementation of the `char_traits` and `iterator_traits` templates.)

For example, here's the Dinkumware `hash_set` declaration (again, tweaked for presentation):

Click here to view code image

```
template<typename T, typename CompareFunction>
class hash_compare;

template<typename T,
         typename HashingInfo = hash_compare<T, less<T> >,
         typename Allocator = allocator<T> >
class hash_set;
```

The interesting part of this interface design is the use of `HashingInfo`. The container's hashing and comparison functions are stored there, but the `HashingInfo` type also holds enums controlling the minimum number of buckets in the table as well as the maximum allowable ratio of container elements to buckets. When this ratio is exceeded, the number of buckets in the table is increased, and some elements in the table are rehashed. (SGI provides member functions that afford similar control over the number of table buckets and, hence, the ratio of table elements to buckets.)

After some tweaks for presentation, `hash_compare` (the default value for `HashingInfo`) looks more or less like this:

Click here to view code image

```
template<typename T, typename CompareFunction = less<T> >
class hash_compare {
public:

  enum {
    bucket_size = 4,                    // max ratio of
elements to buckets
    min_buckets = 8                     // minimum number of
buckets

  };

  size_t operator()(const T&) const;    // hash function

  bool operator()(const T&,             // comparison
function
```

```
                  const T&) const;

  ...                                     // a few things
omitted, including
};                                        // the use of
CompareFunction
```

The overloading of `operator()` (in this case, to implement both the hashing and comparison functions) is a strategy that crops up more frequently than you might imagine. For another application of the same idea, take a look at Item 23.

The Dinkumware design allows you to write your own `hash_compare`-like class (possibly by deriving from `hash_compare` itself), and as long as your class provides definitions for `bucket_size`, `min_buckets`, two `operator()` functions (one taking one argument, one taking two), plus a few things I've left out, you can use it to control the configuration and behavior of a Dinkumware `hash_set` or `hash_multiset`. Configuration control for `hash_map` and `hash_multimap` is similar.

Notice that with both the SGI and the Dinkumware designs, you can leave all the decision-making to the implementations and simply write something like this:

```
hash_set<int> intTable;         // create a hashed set of
ints
```

For this to compile, the hash table must hold an integral type (such as `int`), because the default hashing functions are generally limited to integral types. (SGI's default hashing function is slightly more flexible. Item 50 will tell you where to find all the details.)

Under the hood, the SGI and Dinkumware implementations go their very separate ways. SGI employs a conventional open hashing scheme composed of an array (the buckets) of pointers to singly linked lists of elements. Dinkumware also employs open hashing, but its design is based on a novel data structure consisting of an array of iterators (essentially the buckets) into a doubly linked list of elements, where adjacent pairs of iterators identify the extent of the elements in each bucket. (For details, consult Plauger's aptly titled column, "Hash Tables" [16].)

As a user of these implementations, it's likely you'll be interested in the fact that the SGI implementation stores the table elements in singly linked lists, while the Dinkumware implementation uses a doubly linked list. The difference is noteworthy, because it affects the iterator categories for the two implementations. SGI's hashed containers offer forward iterators, so you forgo the ability to perform reverse iterations; there are no `rbegin` or `rend` member functions in SGI's hashed containers. The iterators for Dinkumware's hashed containers are bidirectional, so they offer both forward and reverse traversals. In terms of memory usage, the SGI design is a bit more parsimonious than that from Dinkumware.

Which design is best for you and your applications? I can't possibly know. Only you can determine that, and this Item hasn't tried to give you enough information to come to a reasonable conclusion. Instead, the goal of this Item is to make sure you know that though the STL itself lacks hashed containers, STL-compatible hashed containers (with varying interfaces, capabilities, and behavioral trade-offs) are not difficult to come by. In the case of the SGI and STLport implementations, you can even come by them for free, because they're available for free download.

Chapter 4. Iterators

At first glance, iterators appear straightforward. Look more closely, however, and you'll notice that the standard STL containers offer four different iterator types: `iterator`, `const_iterator`, `reverse_iterator`, and `const_reverse_iterator`. From there it's only a matter of time until you note that of these four types, only one is accepted by containers in certain forms of `insert` and `erase`. That's when the questions begin. Why four iterator types? What is the relationship among them? Are they interconvertible? Can the different types be mixed in calls to algorithms and STL utility functions? How do these types relate to containers and their member functions?

This chapter answers these questions, and it introduces an iterator type that deserves more notice than it usually gets: `istreambuf_iterator`. If you like the STL, but you're unhappy with the performance of `istream_iterator`s when reading character streams, `istreambuf_iterator` could be just the tool you're looking for.

Item 26: Prefer `iterator` to `const_iterator`, `reverse_iterator`, and `const_reverse_iterator`

As you know, every standard container offers four types of iterator. For a `container<T>`, the type `iterator` acts like a `T*`, while `const_iterator` acts like a `const T*` (which you may also see written as a `T const *`; they mean the same thing). Incrementing an `iterator` or a `const_iterator` moves you to the next element in the container in a traversal starting at the front of the container and proceeding to the back. `reverse_iterator` and `const_reverse_iterator` also act like `T*` and `const T*`, respectively, except that incrementing these iterator types moves you to the next element in the container in a traversal from back to front.

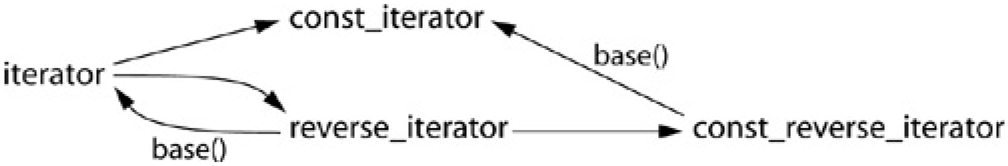
Let me show you two things. First, take a look at some signatures for `insert` and `erase` in `vector<T>`:

Click here to view code image

```
iterator insert(iterator position, const T& x);
iterator erase(iterator position);
iterator erase(iterator rangeBegin, iterator rangeEnd);
```

Every standard container offers functions analogous to these, though the return types vary, depending on the container type. The thing to notice is that these functions demand parameters of type `iterator`. Not `const_iterator`, not `reverse_iterator`, not `const_reverse_iterator`. Always `iterator`. Though containers support four iterator types, one of those types has privileges the others do not have. That type is `iterator`. `iterator` is special.

The second thing I want to show you is this diagram, which displays the conversions that exist among iterator types.



The diagram shows that there are implicit conversions from `iterator` to `const_iterator`, from `iterator` to `reverse_iterator`, and from `reverse_iterator` to `const_reverse_iterator`. It also shows that a `reverse_iterator` may be converted into an `iterator` by using the `reverse_iterator`'s `base` member function, and a `const_reverse_iterator` may similarly be converted into a `const_iterator` via `base`. The diagram does not show that the iterators obtained from `base` may not be the ones you want. For the story on that, turn to Item 28.

You'll observe that there is no way to get from a `const_iterator` to an `iterator` or from a `const_reverse_iterator` to a `reverse_iterator`. This is important, because it means that if you have a `const_iterator` or a `const_reverse_iterator`, you'll find it difficult to use those iterators with some container member functions. Such functions demand `iterator`s, and since there's no conversion path from the const iterator types back to `iterator`, the const iterator types are largely useless if you want to use them to specify insertion positions or elements to be erased.

Don't be fooled into thinking that this means const iterators are useless in general. They're not. They're perfectly useful with algorithms, because algorithms don't usually care what kind of iterators they work with, as long as they are of the appropriate category. const iterators are also acceptable for many container member functions. It's only some forms of `insert` and `erase` that are picky.

I wrote that const iterators are "largely" useless if you want to specify insertion positions or elements to be erased. The implication is that they are not completely useless. That's true. They can be useful if you can find a way to get an `iterator` from a `const_iterator` or from a `const_reverse_iterator`. That's often possible. It isn't *always* possible, however, and even when it is, the way to do it isn't terribly obvious. It may not be terribly efficient, either. The topic is meaty enough to justify its own Item, so turn to Item 27 if you're interested in the details. For now, we have enough information to understand why it often makes sense to prefer `iterator`s to const and reverse iterators:

- Some versions of `insert` and `erase` require `iterator`s. If you want to call those functions, you're going to have to produce `iterator`s. const and reverse iterators won't do.
- It's not possible to implicitly convert a `const_iterator` to an `iterator`, and the technique described in Item 27 for generating an `iterator` from a `const_iterator` is neither universally applicable nor guaranteed to be efficient.
- Conversion from a `reverse_iterator` to an `iterator` may require iterator adjustment after the conversion. Item 28 explains when and why.

All these things conspire to make working with containers easiest, most efficient, and least likely to harbor subtle bugs if you prefer `iterator`s to their const and reverse colleagues.

Practically speaking, you are more likely to have a choice when it comes to `iterator`s and `const_iterator`s. The decision between `iterator` and `reverse_iterator` is often made for you. You either need a front-to-back traversal or a back-to-front traversal, and that's pretty much that. You choose the one you need, and if that means choosing `reverse_iterator`, you choose `reverse_iterator` and use `base` to convert it to an `iterator` (possibly preceded by an offset adjustment—see Item 28) when you want to make calls to container member functions that require `iterator`s.

When choosing between `iterator`s and `const_iterator`s, there are reasons to choose `iterator`s even when you could use a `const_iterator` and when you have no need to use the iterator as a parameter to a container member function. One of the most irksome involves comparisons between `iterator`s and `const_iterator`s. I hope we can all agree that this is reasonable code:

Click here to view code image

```
typedef deque<int> IntDeque;                          // STL
container and
```

```
typedef IntDeque::iterator Iter;                // iterator
types are easier
typedef IntDeque::const_iterator ConstIter;     // to work
with if you
                                                // use some
typedefs
Iter i;
ConstIter ci;

...                                             // make i and
ci point into
                                                // the same
container

if (i == ci) ...                                // compare an
iterator
                                                // and a
const_iterator
```

All we’re doing here is comparing two iterators into a container, the kind of comparison that’s the bread and butter of the STL. The only twist is that one object is of type `iterator` and one is of type `const_iterator`. This should be no problem. The `iterator` should be implicitly converted into a `const_iterator`, and the comparison should be performed between two `const_iterator`s.

With well-designed STL implementations, this is precisely what happens, but with other implementations, the code will not compile. The reason is that such implementations declare `operator==` for `const_iterator`s as a member function instead of as a non-member function, but the cause of the problem is likely to be of less interest to you than the workaround, which is to swap the order of the iterators, like this:

Click here to view code image

```
if (ci == i) ...                                 //
workaround for when the
                                                 // above
won't compile
```

This kind of problem can arise whenever you mix `iterator`s and `const_iterator`s (or `reverse_iterator`s and `const_reverse_iterator`s) in the same expression, not just when you are comparing them. For example, if you try to subtract one random access iterator from another,

```
if (i - ci >= 3) ...                             // if i is
```

```
at least 3 beyond ci ...
```

your (valid) code may be (incorrectly) rejected if the iterators aren't of the same type. The simplest workaround in this case is to turn the `iterator` into a `const_iterator` through a (safe) cast:

Click here to view code image

```
if (static_cast<ConstIter>(i) - ci >= 3) ...     //
workaround for the
                                                 // above
won't compile
```

The easiest way to guard against these kinds of problems is to minimize your mixing of iterator types, and that, in turn, leads back to preferring `iterator`s to `const_iterator`s. From the perspective of `const` correctness (a worthy perspective, to be sure), staying away from `const_iterator`s simply to avoid potential implementation shortcomings (all of which have workarounds) seems unjustified, but in conjunction with the anointed status of `iterator`s in some container member functions, it's hard to avoid the practical conclusion that `const_iterator`s are not only less useful than `iterator`s, sometimes they're just not worth the trouble.

Item 27: Use `distance` and `advance` to convert a container's `const_iterator`s to `iterator`s

Item 26 points out that some container member functions that take iterators as parameters insist on *`iterator`*s; `const_iterator`s won't do. So what do you do if you have a `const_iterator` in hand and you want to, say, insert a new value into a container at the position indicated by the iterator? Somehow you've got to turn your `const_iterator` into an `iterator`, and you have to take an active role in doing it, because, as Item 26 explains, there is no implicit conversion from `const_iterator` to `iterator`.

I know what you're thinking. You're thinking, "When all else fails, get a bigger hammer." In the world of C++, that can mean only one thing: casting. Shame on you for such thoughts. Where do you get these ideas?

Let us confront this cast obsession of yours head on. Look what happens when you try to cast a `const_iterator` to an `iterator`:

Click here to view code image

```
typedef deque<int> IntDeque;                    //
```

```
convenience typedefs
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;

ConstIter ci;                                   // ci is a
const_iterator

...

Iter i(ci);                          // error! no implicit
conversion from
                                     // const_iterator to
iterator

Iter i(const_cast<Iter>(ci));        // still an error! can't
cast a
                                     // const_iterator to an
iterator
```

This example happens to use `deque`, but the outcome would be the same for `list`, `set`, `multiset`, `map`, `multimap`, and the hashed containers described in Item 25. The line using the cast *might* compile in the case of `vector` or `string`, but those are special cases we'll consider in a moment.

The reason the cast won't compile is that for these container types, `iterator` and `const_iterator` are different classes, barely more closely related to one another than `string` and `complex<double>`. Trying to cast one type to the other is nonsensical, and that's why the `const_cast` is rejected. `static_cast`, `reinterpret_cast`, and a C-style cast would lead to the same end.

Alas, the cast that won't compile might compile if the iterators' container were a `vector` or a `string`. That's because it is common for implementations of these containers to use pointers as iterators. For such implementations, `vector<T>::iterator` is a typedef for `T*`, `vector<T>::const_iterator` is a typedef for `const T*`, `string::iterator` is a typedef for `char*`, and `string::const_iterator` is a typedef for `const char*`. With such implementations, the `const_cast` from a `const_iterator` to an `iterator` will compile and do the right thing, because the cast is converting a `const T*` into a `T*`. Even under such implementations, however, `reverse_iterator`s and `const_reverse_iterator`s are true classes, so you can't `const_cast` a `const_reverse_iterator` to a `reverse_iterator`. Also, as Item 50 explains, even implementations where `vector` and `string` iterators are pointers might use that representation only when compiling in release mode. All these factors lead to the conclusion that casting `const` iterators to iterators is ill-advised even for `vector` and `string`, because its portability is doubtful.

If you have access to the container a `const_iterator` came from, there is a safe, portable way to get its corresponding `iterator`,[†] and it involves no circumvention of the type system. Here's the essence of the solution, though it must be modified slightly before it will compile:

Click here to view code image

```
typedef deque<int> IntDeque;                    // as before
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;

IntDeque d;

ConstIter ci;
...                                  // make ci point into d

Iter i(d.begin());                   // initialize i to
d.begin()

advance(i, distance(i, ci));         // move i up to where ci
is
                                     // (but see below for
why this must
                                     // be tweaked before it
will compile)
```

This approach is so simple and direct, it's startling. To get an `iterator` pointing to the same container element as a `const_iterator`, create a new `iterator` at the beginning of the container, then move it forward until it's as far from the beginning of the container as the `const_iterator` is! This task is facilitated by the function templates `advance` and `distance`, both of which are declared in `<iterator>`. `distance` reports how far apart two iterators into the same container are, and `advance` moves an iterator a specified distance. When `i` and `ci` point into the same container, the expression `advance(i, distance(i, ci))` makes `i` and `ci` point to the same place in the container.

[†] I have since discovered that the approach described here may fail for `string` implementations that use reference counting. For details, consult jep's 8/22/01 comment regarding page 121 of this book at http://www.aristeia.com/BookErrata/estl1e-errata.html.

Well, it would if it would compile, but it won’t. To see why, look at the declaration for `distance`:

```
template <typename InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

Don’t get hung up on the fact that the return type of the function is 56 characters long and mentions dependent types like `difference_type`. Instead, focus your attention on the uses of the type parameter `InputIterator`:

Click here to view code image

```
template <typename InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

When faced with a call to `distance`, your compilers must deduce the type represented by `InputIterator` by examining the arguments used in the call. Look again at the call to `distance` in the code I said wasn’t quite right:

```
advance(i, distance(i, ci));             // move i up to
where ci is
```

Two parameters are passed to `distance`, `i` and `ci`. `i` is of type `Iter`, which is a typedef for `deque<int>::iterator`. To compilers, that implies that `InputIterator` in the call to `distance` is `deque<int>::iterator`. `ci`, however, is of type `ConstIter`, which is a typedef for `deque<int>::const_iterator`. *That* implies that `InputIterator` is of type `deque<int>::const_iterator`. It’s not possible for `InputIterator` to be two different types at the same time, so the call to `distance` fails, typically yielding some long-winded error message that may or may not indicate that the compiler couldn’t figure out what type `InputIterator` is supposed to be.

To get the call to compile, you must eliminate the ambiguity. The easiest way to do that is to explicitly specify the type parameter to be used by `distance`, thus obviating the need for your compilers to figure it out for themselves:

Click here to view code image

```
advance(i, distance<ConstIter>(i, ci));  // figure the
distance between
                                         // i and ci (as
const_iterators),
                                         // then move i that
distance
```

We now know how to use `advance` and `distance` to get an `iterator` corresponding to a `const_iterator`, but we have so far sidestepped a question of considerable practical interest: How efficient is this technique? The answer is simple. It’s as efficient as the iterators allow it to be. For random access iterators (such as those sported by `vector`, `string`, and `deque`), it’s a constant-time operation. For bidirectional iterators (i.e., those for all other standard containers and for some implementations of the hashed containers (see Item 25)), it’s a linear-time operation.

Because it may take linear time to produce an `iterator` equivalent to a `const_iterator`, you may wish to rethink designs that require producing `iterator`s from `const_iterator`s. Such a consideration, in fact, helps motivate Item 26, which advises you to prefer `iterator`s over `const` and reverse iterators when dealing with containers.

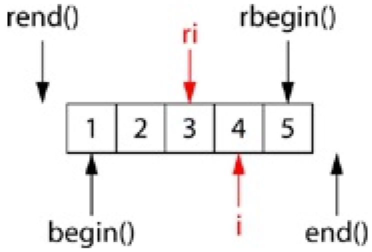
Item 28: Understand how to use a `reverse_iterator`’s base `iterator`

Invoking the `base` member function on a `reverse_iterator` yields the “corresponding” `iterator`, but it’s not really clear what that means. As an example, take a look at this code, which puts the numbers 1-5 in a vector, sets a `reverse_iterator` to point to the 3, and initializes an `iterator` to the `reverse_iterator`’s base:

Click here to view code image

```
vector<int> v;
v.reserve(5);                          // see Item 14

for (int i = 1; i <= 5; ++i) {         // put 1-5 in the
vector
  v.push_back(i);
}

vector<int>::reverse_iterator ri =     // make ri point to
the 3
  find(v.rbegin(), v.rend(), 3);

vector<int>::iterator i(ri.base());    // make i the same as
ri's base
```

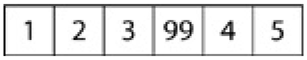
After executing this code, things can be thought of as looking like this:



This picture is nice, displaying the characteristic offset of a `reverse_iterator` and its corresponding base `iterator` that mimics the offset of `rbegin()` and `rend()` with respect to `begin()` and `end()`, but it doesn't tell you everything you need to know. In particular, it doesn't explain how to use `i` to perform operations you'd like to perform on `ri`.

As Item 26 explains, some container member functions accept only `iterator`s as iterator parameters, so if you want to, say, insert a new element at the location identified by `ri`, you can't do it directly; `vector`'s `insert` function won't take `reverse_iterator`s. You'd have a similar problem if you wanted to erase the element pointed to by `ri`. The `erase` member functions reject `reverse_iterator`s, insisting instead on `iterator`s. To perform insertions or erasures, you must convert `reverse_iterator`s into `iterator`s via `base`, then use the `iterator`s to get the jobs done.

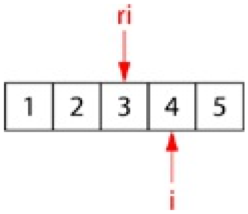
So let's suppose you *do* want to insert a new element into `v` at the position indicated by `ri`. In particular, let's assume you want to insert the value 99. Bearing in mind that `ri` is part of a traversal from right to left in the picture above and that insertion takes place in *front* of the element indicated by the iterator used to specify the insertion position, we'd expect the 99 to end up in front of the 3 with respect to a reverse traversal. After the insertion, then, `v` would look like this:



Of course, we can't use `ri` to indicate where to insert something, because it's not an `iterator`. We must use `i` instead. As noted above, when `ri` points at 3, `i` (which is `ri.base()`) points at 4. That's exactly where `i` needs to point for an insertion if the inserted value is to end up where it would have had we been able to use `ri` to specify the insertion location. Conclusion?

- To emulate insertion at a position specified by a `reverse_iterator ri`, insert at the position `ri.base()` instead. For purposes of insertion, `ri` and `ri.base()` are equivalent, and `ri.base()` is truly the `iterator` *corresponding* to `ri`.

Let us now consider erasing an element. Look again at the relationship between `ri` and `i` in the original vector (i.e., prior to the insertion of 99):



If we want to erase the element pointed to by `ri`, we can't just use `i`, because `i` doesn't point to the same element as `ri`. Instead, we must erase the element *preceding* `i`. Hence,

- To emulate erasure at a position specified by a `reverse_iterator ri`, erase at the position *preceding* `ri.base()` instead. For purposes of erasure, `ri` and `ri.base()` are *not* equivalent, and `ri.base()` is *not* the `iterator` corresponding to `ri`.

It's worth looking at the code to perform such an erasure, because it holds a surprise.

[Click here to view code image]

```
vector<int> v;

...                                          // as above, put
1-5 in v

vector<int>::reverse_iterator ri =           // as above,
make ri point to the 3
  find(v.rbegin(), v.rend(), 3);

v.erase(--ri.base());                        // attempt to
erase at the position
                                             // preceding
ri.base(); for a vector,
                                             // this will
typically not compile
```

There's nothing wrong with this design. The expression `--ri.base()` correctly specifies the element we'd like to erase. Furthermore, this code will work with every standard container except `vector` and `string`. It might work with `vector` and `string`, too, but for many `vector` and `string` implementations, it won't compile. In such implementations, `iterator`s (and `const_iterator`s) are implemented as built-in pointers, so the result of `ri.base()` is a pointer.

Both C and C++ dictate that pointers returned from functions shall not be modified, so for STL platforms where `string` and `vector` `iterator`s are pointers, expressions like `--ri.base()` won't compile. To portably erase something at a position specified by a `reverse_iterator`, then, you must take pains to avoid modifying `base`'s return value. No problem. If you can't decrement the result of calling `base`, just increment the `reverse_iterator` and *then* call `base`!

Click here to view code image

```
...                                          // as above

v.erase((++ri).base());                      // erase the
element pointed to by
                                             // ri; this
should always compile
```

Because this approach works with every standard container, it is the preferred technique for erasing an element pointed to by a `reverse_iterator`.

It should now be clear that it's not accurate to say that a `reverse_iterator`'s `base` member function returns the "corresponding" `iterator`. For insertion purposes, it does, but for erasure purposes, it does not. When converting `reverse_iterator`s to `iterator`s, it's important that you know what you plan to do with the resulting `iterator`, because only then can you determine whether the `iterator` you have is the one you need.

Item 29: Consider `istreambuf_iterator`s for character-by-character input

Let's suppose you'd like to copy a text file into a `string` object. This seems like a pretty reasonable way to do it:

Click here to view code image

```
ifstream inputFile("interestingData.txt");

string fileData((istream_iterator<char>(inputFile)), // read
inputFile into
                 istream_iterator<char>());          //
fileData; see below
                                                     // for
```

```
why this isn't
                                                     // quite
right, and see
                                                     // Item
6 for a warning
                                                     // about
this syntax
```

It wouldn’t take long before you’d notice that this approach fails to copy whitespace in the file into the string. That’s because `istream_iterator`s use `operator>>` functions to do the actual reading, and, by default, `operator>>` functions skip whitespace.

Assuming you’d like to retain the whitespace, all you need to do is override the default. Just clear the `skipws` flag for the input stream:

Click here to view code image

```
ifstream inputFile("interestingData.txt");

inputFile.unsetf(ios::skipws);                  // disable
the skipping of
                                                //
whitespace in inputFile

string fileData((istream_iterator<char>(inputFile)),
                 istream_iterator<char>());
```

Now all the characters in `inputFile` are copied into `fileData`.

Alas, you may discover that they aren’t copied as quickly as you’d like. The `operator>>` functions on which `istream_iterator`s depend perform formatted input, and that means they must undertake a fair amount of work on your behalf each time you call one. They have to create and destroy `sentry` objects (special iostream objects that perform setup and cleanup activities for each call to `operator>>`), they have to check stream flags that might affect their behavior (e.g., `skipws`), they have to perform comprehensive checking for read errors, and, if they encounter a problem, they have to check the stream’s exception mask to determine whether an exception should be thrown. Those are all important activities if you’re performing formatted input, but if all you want to do is grab the next character from the input stream, it’s overkill.

A more efficient approach is to use one of the STL’s best kept secrets: `istreambuf_iterator`s. You use `istreambuf_iterator`s like `istream_iterator`s, but where `istream_iterator<char>` objects use `operator>>` to read individual characters from an input stream, `istreambuf_iterator<char>` objects go straight to the stream's buffer and read the next character directly. (More specifically, an `istreambuf_iterator<char>` object reading from an istream `s` will call `s.rdbuf()->sgetc()` to read `s`'s next character.)

Modifying our file-reading code to use `istreambuf_iterator`s is so easy, most Visual Basic programmers need no more than two tries to get it right:

Click here to view code image

```
ifstream inputFile("interestingData.txt");

string fileData((istreambuf_iterator<char>(inputFile)),
                 istreambuf_iterator<char>());
```

Notice how there's no need to "unset" the `skipws` flag here. `istreambuf_iterator`s never skip any characters. Whatever's next in the stream buffer, that's what they grab.

Compared to `istream_iterator`s, they grab it quickly—up to 40% faster in the simple benchmarks I performed, though don't be surprised if your mileage varies. Don't be surprised if the speed advantage increases over time, too, because `istreambuf_iterator`s inhabit a seldom-visited corner of the STL where implementers haven't yet spent a lot of time on optimizations. For example, in one implementation I used, `istreambuf_iterator`s were only about 5% faster than `istream_iterator`s on my primitive tests. Such implementations clearly have lots of room to streamline their `istreambuf_iterator` implementations.

If you need to read the characters in a stream one by one, you don't need the power of formatted input, and you care about how long it takes to read the stream, typing three extra characters per iterator is a small price to pay for what is often a significant increase in performance. For unformatted character-by-character input, you should always consider `istreambuf_iterator`s.

While you're at it, you should also consider `ostreambuf_iterator`s for the corresponding unformatted character-by-character output operations. They avoid the overhead (and flexibility) of their `ostream_iterator` cousins, so they generally outperform them, too.

Chapter 5. Algorithms

I noted at the beginning of Chapter 1 that containers get the lion's share of the STL acclaim. In a sense, that's understandable. The containers are remarkable accomplishments, and they make life easier for legions of C++ programmers on a daily basis. Still, the STL algorithms are significant in their own right, equally capable of lightening a developer's burden. In fact, given that there are over 100 algorithms, it's easy to argue that they offer programmers a more finely honed tool set than the containers (a mere eight strong) could ever hope to match. Perhaps their number is part of the problem. Making sense of eight distinct container types is surely less work than remembering 70 algorithm names and trying to keep track of which does what.

I have two primary goals in this chapter. First, I want to introduce you to some lesser-known algorithms by showing you how they can make your life easier. Rest assured that I'm not going to punish you with lists of names to memorize. The algorithms I show you are in this chapter because they solve common problems, like performing case-insensitive string comparisons, efficiently finding the *n* most desirable objects in a container, summarizing the characteristics of all the objects in a range, and implementing the behavior of `copy_if` (an algorithm from the original HP STL that was dropped during standardization).

My second goal is to show you how to avoid common usage problems with the algorithms. You can't call `remove`, for example, or its cousins `remove_if` and `unique` unless you understand *exactly* what these algorithms do (and do *not* do). This is especially true when the range from which you're `remove`ing something holds pointers. Similarly, a number of algorithms work only with sorted ranges, so you need to understand which ones they are and why they impose that constraint. Finally, one of the most common algorithm-related mistakes involves asking an algorithm to write its results to a place that doesn't exist, so I explain how this absurdity can come about and how to ensure that you're not afflicted.

By the end of the chapter, you may not hold algorithms in the same high regard you probably already accord containers, but I'm hopeful you'll be willing to let them share the limelight more often than you have in the past.

Item 30: Make sure destination ranges are big enough

STL containers automatically expand themselves to hold new objects as they are added (via `insert`, `push_front`, `push_back`, etc.). This works so well, some programmers lull themselves into the belief that they never have to worry about making room for objects in containers, because the containers themselves take care of things. If only it were so!

The problems arise when programmers *think* about inserting objects into containers, but don't tell the STL what they're thinking. Here's a common way this can manifest itself:

Click here to view code image

```
int transmogrify(int x);                     // this function
produces
                                             // some new
value from x

vector<int> values;

...                                          // put data into
values

vector<int> results;                         // apply
transmogrify to
transform(values.begin(), values.end(),      // each object
in values,
          results.end(),                     // appending the
return
          transmogrify);                     // values to
results; this
                                             // code has a
bug!
```

In this example, `transform` is told that the beginning of its destination range is `results.end()`, so that's where it starts writing the results of invoking `transmogrify` on every element of `values`. Like every algorithm that uses a destination range, `transform` writes its results by making assignments to the elements in the destination range. `transform` will thus apply `transmogrify` to `values[0]` and assign the result to `*results.end()`. It will then apply `transmogrify` to `values[1]` and assign the result to `*(results.end()+1)`. This can lead only to disaster, because *there is no object* at `*results.end()`, much less at `*(results.end()+1)`! The call to `transform` is wrong, because it’s asking for assignments to be made to objects that don’t exist. (Item 50 explains how a debugging implementation of the STL can detect this problem at runtime.)

Programmers who make this kind of mistake almost always intend for the results of the algorithm they’re calling to be `insert`ed into the destination container. If that’s what you want to happen, you have to say so. The STL is a library, not a psychic. In this example, the way to say “please put `transform`’s results at the end of the container called `results`” is to call `back_inserter` to generate the iterator specifying the beginning of the destination range:

Click here to view code image

```
vector<int> results;                        // apply
transmogrify to
transform(values.begin(), values.end(),     // each object in
values,
          back_inserter(results),           // inserting the
return
          transmogrify);                    // values at the
end of
                                            // results
```

Internally, the iterator returned by `back_inserter` causes `push_back` to be called, so you may use `back_inserter` with any container offering `push_back` (i.e., any of the standard sequence containers: `vector`, `string`, `deque`, and `list`). If you’d prefer to have an algorithm insert things at the front of a container you can use `front_inserter`. Internally, `front_inserter` makes use of `push_front`, so `front_inserter` works only for the containers offering that member function (i.e, `deque` and `list`):

Click here to view code image

```
...                                         // same as before

list<int> results;                          // results is now
a list

transform(values.begin(), values.end(),     // insert
transform's
          front_inserter(results),          // results at the
front of
          transmogrify);                    // results in
```

```
reverse order
```

Because `front_inserter` causes each object added to `results` to be `push_front`ed, the order of the objects in `results` will be the *reverse* of the order of the corresponding objects in `values`. This is one reason why `front_inserter` isn't used as often as `back_inserter`. Another reason is that `vector` doesn't offer `push_front`, so `front_inserter` can't be used with `vector`s.

If you want `transform` to put its output at the front of `results`, but you also want the output to be in the same order as the corresponding objects in `values`, just iterate over `values` in reverse order:

Click here to view code image

```
list<int> results;                              // same as
before

transform(values.rbegin(), values.rend(),       // insert
transform's
          front_inserter(results),              // results at
the front of
          transmogrify);                        // results;
preserve the
                                                // relative
object ordering
```

Given that `front_inserter` lets you force algorithms to insert their results at the front of a container and `back_inserter` lets you tell them to put their results at the back of a container, it's little surprise that `inserter` allows you to force algorithms to insert their results into containers at arbitrary locations:

Click here to view code image

```
vector<int> values;                              // as before
...

vector<int> results;                             // as before,
except now
...                                              // results
has some data
                                                 // in it
prior to the call to
                                                 // transform

transform(values.begin(),
values.end(),                              // insert the
```

```
          inserter(results, results.begin() + results.size()
/ 2), // results of
          transmogrify);
the trans-

mogrifica-

tions at

the middle

of results
```

Regardless of whether you use `back_inserter`, `front_inserter`, or `inserter`,[†] each insertion into the destination range is done one object at a time. Item 5 explains that this can be expensive for contiguous-memory containers (`vector`, `string`, and `deque`), but Item 5's suggested solution (using range member functions) can't be applied when it's an algorithm doing the inserting. In this example, `transform` will write to its destination range one value at a time, and there's nothing you can do to change that.

When the container into which you're inserting is a `vector` or a `string`, you can minimize the expense by following the advice of Item 14 and calling `reserve` in advance. You'll still have to absorb the cost of shifting elements up each time an insertion takes place, but at least you'll avoid the need to reallocate the container's underlying memory:

Click here to view code image

```
vector<int> values;                                // as
above
vector<int> results;
...

results.reserve(results.size() + values.size());   // ensure
that results has
                                                   // the
capacity for at least
                                                   //
```

[†] Unlike `back_inserter` and `front_inserter`, which just call `push_back` and `push_front`, respectively, `inserter` returns an iterator that keeps track of its most recent insertion location. Subsequent insertions take place following the most recently inserted element.

```
values.size() more
                                                    //
elements

transform(values.begin(),
values.end(),                             // as above,
          inserter(results, results.begin() + results.size()
/ 2), // but results
          transmogrify);
won't do

any reallo-

cations
```

When using `reserve` to improve the efficiency of a series of insertions, always remember that `reserve` increases only a container's *capacity*; the container's size remains unchanged. Even after calling `reserve`, you must use an insert iterator (e.g., one of the iterators returned from `back_inserter`, `front_inserter`, or `inserter`) with an algorithm when you want that algorithm to add new elements to a `vector` or `string`.

To make this absolutely clear, here is the *wrong* way to improve the efficiency of the example at the beginning of this Item (the one where we append to `results` the outcome of `transmogrify`ing the data in `values`):

Click here to view code image

```
vector<int> values;                                  // as
above
vector<int> results;
...

results.reserve(results.size() + values.size());     // as
above

transform(values.begin(), values.end(),              // write
the results of
          results.end(),                             // the
transmogrifications
          transmogrify);                             // to
uninitialized memory;
                                                     //
behavior is undefined!
```

In this code, `transform` blithely attempts to make assignments to the raw, uninitialized memory at the end of `results`. In general, this will fail at runtime, because assignment is an operation that makes sense only between two objects, not between one object and a chunk of primordial bits. Even if this code happens to do what you want it to, `results` won't know about the new "objects" `transform` "created" in its unused capacity. As far as `results` would be aware, its size would be the same after the call to `transform` as it was before. Similarly, its end iterator would point to the same place it did prior to the call to `transform`. Conclusion? Using `reserve` without also using an insert iterator leads to undefined behavior inside algorithms as well as to corrupted containers.

The correct way to code this example uses both `reserve` *and* an insert iterator:

Click here to view code image

```
vector<int> values;                                 // as
above
vector<int> results;
...

results.reserve(results.size() + values.size());    // as
above

transform(values.begin(), values.end(),             // write
the results of
          back_inserter(results),                   // the
transmogrifications
          transmogrify);                            // to the
end of results,
                                                    //
avoiding reallocations
                                                    // during
the process
```

So far, I've assumed that you want algorithms like `transform` to insert their results as new elements into a container. This is a common desire, but sometimes you want to overwrite the values of existing container elements instead of inserting new ones. When that's the case, you don't need an insert iterator, but you still need to follow the advice of this Item and make sure your destination range is big enough.

For example, suppose you want `transform` to overwrite `results`' elements. As long as `results` has at least as many elements as `values` does, that's easy. If it doesn't, you must either use `resize` to make sure it does,

Click here to view code image

```
vector<int> values;
vector<int> results;
...

if (results.size() < values.size()) {              // make sure
results is at
    results.resize(values.size());                 // least as
big as values is
}

transform(values.begin(), values.end(),            // overwrite
the first
          results.begin(),                         //
values.size() elements of
          transmogrify);                           // results
```

or you can `clear` `results` and then use an insert iterator in the usual fashion:

Click here to view code image

```
...

results.clear();                               // destroy all
elements in
                                               // results

results.reserve(values.size());                // reserve
enough space

transform(values.begin(), values.end(),        // put
transform's return
          back_inserter(results),              // values into
results
          transmogrify);
```

This Item has demonstrated a number of variations on a theme, but I hope the underlying melody is what sticks in your mind. Whenever you use an algorithm requiring specification of a destination range, ensure that the destination range is big enough already or is increased in size as the algorithm runs. To increase the size as you go, use insert iterators, such as `ostream_iterator`s or those returned by `back_inserter`, `front_inserter`, or `inserter`. That's all you need to remember.

Item 31: Know your sorting options

How can I sort thee? Let me count the ways.

When many programmers think of ordering objects, only a single algorithm comes to mind: `sort`. (Some programmers think of `qsort`, but once they've read Item 46, they recant and replace thoughts of `qsort` with those of `sort`.)

Now, `sort` is a wonderful algorithm, but there's no point in squandering wonder where you don't need it. Sometimes you don't need a full sort. For example, if you have a vector of `Widget`s and you'd like to select the 20 highest-quality `Widget`s to send to your most loyal customers, you need to do only enough sorting to identify the 20 best `Widget`s; the remainder can remain unsorted. What you need is a partial sort, and there's an algorithm called `partial_sort` that does exactly what the name suggests:

Click here to view code image

```
bool qualityCompare(const Widget& lhs, const Widget& rhs)
{
  // return whether lhs's quality is greater than rhs's
quality
}

...

partial_sort(widgets.begin(),                // put the best
20 elements
             widgets.begin() + 20,           // (in order) at
the front of
             widgets.end(),                  // widgets
             qualityCompare);

...                                          // use
widgets...
```

After the call to `partial_sort`, the first 20 elements of `widgets` are the best in the container and are in order, i.e., the highest-quality `Widget` is `widgets[0]`, the next highest is `widgets[1]`, etc. That makes it easy to send your best `Widget` to your best customer, the next best `Widget` to your next best customer, etc.

If all you care about is that the 20 best `Widget`s go to your 20 best customers, but you don't care which `Widget` goes to which customer, `partial_sort` gives you more than you need. In that case, all you need is the 20 best `Widget`s in *any* order. The STL has an algorithm that does exactly what you want, though the name isn't likely to spring to mind. It's called `nth_element`.

`nth_element` sorts a range so that the element at position *n* (which you specify) is the one that would be there if the range had been fully sorted. In addition, when `nth_element` returns, none of the elements in the positions up to *n* follow the element at position *n* in the sort order, and none of the elements in positions following *n* precede the element at position *n* in the sort order. If that sounds complicated, it's only because I have to select my words carefully. I'll explain why in a moment, but first let's look at how to use `nth_element` to make sure the best 20 `Widget`s are at the front of the `widgets` vector:

Click here to view code image

```
nth_element(widgets.begin(),                  // put the best 20
elements
            widgets.begin() + 19,             // at the front of
widgets,
            widgets.end(),                    // but don't worry
about
            qualityCompare);                  // their order
```

As you can see, the call to `nth_element` is essentially identical to the call to `partial_sort`. The only difference in their effect is that `partial_sort` sorts the elements in positions 1-20, while `nth_element` doesn't. Both algorithms, however, move the 20 highest-quality `Widget`s to the front of the vector.

That gives rise to an important question. What do these algorithms do when there are elements with the same level of quality? Suppose, for example, there are 12 elements with a quality rating of 1 (the best possible) and 15 elements with a quality rating of 2 (the next best). In that case, choosing the 20 best `Widget`s involves choosing the 12 with a rating of 1 and 8 of the 15 with a rating of 2. How should `partial_sort` and `nth_element` determine which of the 15 to put in the top 20? For that matter, how should `sort` figure out which order to put elements in when multiple elements have equivalent values?

`partial_sort` and `nth_element` order elements with equivalent values any way they want to, and you can't control this aspect of their behavior. (See Item 19 for what it means for two values to be equivalent.) In our example, when faced with the need to choose `Widget`s with a quality rating of 2 to put into the last 8 spots in the vector's top 20, they'd choose whichever ones they wanted. That's not unreasonable. If you ask for the 20 best `Widget`s and some `Widget`s are equally good, you're in no position to complain as long as the 20 you get back are at least as good as the ones you didn’t.

For a full sort, you have slightly more control. Some sorting algorithms are *stable*. In a stable sort, if two elements in a range have equivalent values, their relative positions are unchanged after sorting. Hence, if `Widget` A precedes `Widget` B in the (unsorted) `widgets` vector and both have the same quality rating, a stable sorting algorithm will guarantee that after the vector is sorted, `Widget` A still precedes `Widget` B. An algorithm that is not stable would not make this guarantee.

`partial_sort` is not stable. Neither is `nth_element`. `sort`, too, fails to offer stability, but there is an algorithm, `stable_sort`, that does what its name suggests. If you need stability when you sort, you’ll probably want to use `stable_sort`. The STL does not contain stable versions of `partial_sort` or `nth_element`.

Speaking of `nth_element`, this curiously named algorithm is remarkably versatile. In addition to letting you find the top *n* elements of a range, it can also be used to find the median value in a range or to find the value at a particular percentile:

Click here to view code image

```
vector<Widget>::iterator begin(widgets.begin());    //
convenience vars
vector<Widget>::iterator end(widgets.end());        // for
widgets' begin
                                                    // and
end iterators

vector<Widget>::iterator goalPosition;       // iter
indicating where
                                             // the widget of
interest
                                             // is located

// The following code finds the Widget with
// the median level of quality

goalPosition = begin + widgets.size() / 2;   // the widget of
interest
                                             // would be in
the middle
                                             // of the sorted
```

```
vector

nth_element(begin, goalPosition, end,        // find the
median quality
            qualityCompare);                 // value in
widgets

...                                          // goalPosition
now points
                                             // to the Widget
with a
                                             // median level
of quality

// The following code finds the Widget with
// a level of quality at the 75th percentile

vector<Widget>::size_type goalOffset =       // figure out
how far from
  0.25 * widgets.size();                     // the beginning
the
                                             // Widget of
interest is

nth_element(begin, begin + goalOffset, end,  // find the
quality value at
            qualityCompare);                 // the 75th
percentile

...                                          //
begin+goalOffset now
                                             // points to the
Widget
                                             // with the 75th
percentile
                                             // level of
quality
```

`sort`, `stable_sort`, and `partial_sort` are great if you really need to put things in order, and `nth_element` fills the bill when you need to identify the top *n* elements or the element at a particular position, but sometimes you need something similar to `nth_element`, but not quite the same. Suppose, for example, you didn't need to identify the 20 highest-quality `Widget`s. Instead, you needed to identify all the `Widget`s with a quality rating of 1 or 2. You could, of course, sort the vector by quality and then search for the first one with a quality rating worse than 2. That would identify the beginning of a range of poor-quality `Widget`s.

A full sort can be a lot of work, however, and that much work is not necessary for this job. A better strategy is to use the `partition` algorithm, which reorders elements in a range so that all elements satisfying a particular criterion are at the beginning of the range.

For example, to move all the `Widget`s with a quality rating of 2 or better to the front of `widgets`, we define a function that identifies which `Widget`s make the grade,

Click here to view code image

```
bool hasAcceptableQuality(const Widget& w)
{
  // return whether w has a quality rating of 2 or better;
}
```

then pass that function to `partition`:

Click here to view code image

```
vector<Widget>::iterator goodEnd =      // move all widgets
satisfying
  partition(widgets.begin(),            //
hasAcceptableQuality to
            widgets.end(),              // the front of
widgets, and
            hasAcceptableQuality);      // return an iterator
to the first
                                        // widget that isn't
satisfactory
```

After this call, the range from `widgets.begin()` to `goodEnd` holds all the `Widget`s with a quality of 1 or 2, and the range from `goodEnd` to `widgets.end()` contains all the `Widget`s with lower quality ratings. If it were important to maintain the relative positions of `Widget`s with equivalent quality levels during the partitioning, we'd naturally reach for `stable_partition` instead of `partition`.

The algorithms `sort`, `stable_sort`, `partial_sort`, and `nth_element` require random access iterators, so they may be applied only to `vector`s, `string`s, `deque`s, and arrays. It makes no sense to sort elements in standard associative containers, because such containers use their comparison functions to remain sorted at all times. The only container where we might like to use `sort`, `stable_sort`, `partial_sort`, or `nth_element`, but can't, is `list`, and `list` compensates somewhat by offering its `sort` member function. (Interestingly, `list::sort` performs a stable sort.) If you want to sort a `list`, then, you can, but if you want to use `partial_sort`, or `nth_element` on the objects in a `list`, you have to do it indirectly. One indirect approach is to copy the elements into a container with random access iterators, then apply the desired algorithm to that. Another is to create a container of `list::iterator`s, use the algorithm on that container, then access the list elements via the iterators. A third is to use the information in an ordered container of iterators to iteratively `splice` the list's elements into the positions you'd like them to be in. As you can see, there are lots of options.

`partition` and `stable_partition` differ from `sort`, `stable_sort`, `partial_sort`, and `nth_element` in requiring only bidirectional iterators. You can therefore use `partition` and `stable_partition` with any of the standard sequence containers.

Let's summarize your sorting options.

- If you need to perform a full sort on a `vector`, `string`, `deque`, or array, you can use `sort` or `stable_sort`.
- If you have a `vector`, `string`, `deque`, or array and you need to put only the top *n* elements in order, `partial_sort` is available.
- If you have a `vector`, `string`, `deque`, or array and you need to identify the element at position *n* or you need to identify the top *n* elements without putting them in order, `nth_element` is at your beck and call.
- If you need to separate the elements of a standard sequence container or an array into those that do and do not satisfy some criterion, you're probably looking for `partition` or `stable_partition`.
- If your data is in a `list`, you can use `partition` and `stable_partition` directly, and you can use `list::sort` in place of `sort` and `stable_sort`. If you need the effects offered by `partial_sort` or `nth_element`, you'll have to approach the task indirectly, but there are a number of options, as I sketched above.

In addition, you can keep things sorted at all times by storing your data in a standard associative container. You might also consider the standard non-STL container `priority_queue`, which also keeps its elements ordered all the time. (`priority_queue` is traditionally considered part of the STL, but, as I noted in the Introduction, my definition of "the STL" requires that STL containers support iterators, and `priority_queue` doesn’t do iterators.)

“But what about performance?”, you wonder. Excellent question. Broadly speaking, algorithms that do more work take longer to do it, and algorithms that must sort stably take longer than algorithms that can ignore stability. We can order the algorithms we’ve discussed in this Item as follows, with algorithms that tend to use fewer resources (time and space) listed before those that require more:

1. `partition`
2. `stable_partition`
3. `nth_element`
4. `partial_sort`
5. `sort`
6. `stable_sort`

My advice on choosing among the sorting algorithms is to make your selection based on what you need to accomplish, not on performance considerations. If you choose an algorithm that does only what you need to do (e.g., a `partition` instead of a full `sort`), you’re likely to end up with code that’s not only the clearest expression of what you want to do, it’s also the most efficient way to accomplish it using the STL.

Item 32: Follow `remove`-like algorithms by `erase` if you really want to remove something

I begin this Item with a review of `remove`, because `remove` is the most confusing algorithm in the STL. Misunderstanding `remove` is easy, and it’s important to dispel all doubt about what `remove` does, why it does it, and how it goes about doing it.

Here is the declaration for `remove`:

```
template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator
last,
                       const T& value);
```

Like all algorithms, `remove` receives a pair of iterators identifying the range of elements over which it should operate. It does not receive a container, so `remove` doesn’t know which container holds the elements it’s looking at. Furthermore, it’s not possible for `remove` to discover that container, because there is no way to go from an iterator to the container corresponding to that iterator.

Think for a moment about how one eliminates elements from a container.

The only way to do it is to call a member function on that container, almost always some form of `erase`. (`list` has a couple of member functions that eliminate elements and are not named `erase`, but they're still member functions.) Because the only way to eliminate an element from a container is to invoke a member function on that container, and because `remove` cannot know the container holding the elements on which it is operating, *it is not possible for* `remove` *to eliminate elements from a container*. That explains the otherwise baffling observation that `remove`ing elements from a container never changes the number of elements in the container:

Click here to view code image

```
vector<int> v;                          // create a
vector<int> and fill it with
v.reserve(10);                          // the values 1-10.
(See Item 14 for an
for (int i = 1; i <= 10; ++i) {         // explanation of the
reserve call.)
  v.push_back(i);
}

cout << v.size();                       // prints 10

v[3] = v[5] = v[9] = 99;                // set 3 elements to
99

remove(v.begin(), v.end(), 99);         // remove all elements
with value 99

cout << v.size();                       // still prints 10!
```

To make sense of this example, memorize the following:

> `remove` doesn't "really" remove anything, because it can't.

Repetition is good for you:

> `remove` doesn't "really" remove anything, *because it can't*.

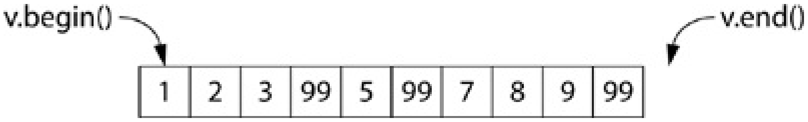
`remove` doesn't know the container it's supposed to remove things from, and without that container, there's no way for it to call the member functions that are necessary if one is to "really" remove something.

That explains what `remove` doesn't do, and it explains why it doesn't do it.

What we need to review now is what remove *does* do.

Very briefly, `remove` moves elements in the range it's given until all the "unremoved" elements are at the front of the range (in the same relative order they were in originally). It returns an iterator pointing one past the last "unremoved" element. This return value is the "new logical end" of the range.

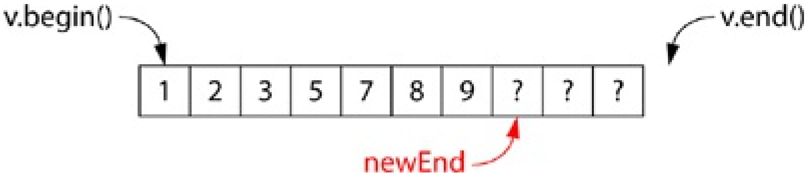
In terms of our example, this is what `v` looks like prior to calling `remove`,



and if we store `remove`'s return value in a new iterator called `newEnd`,

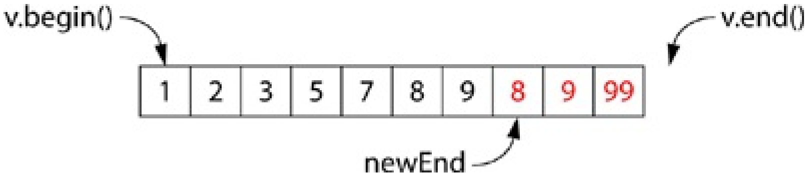
```
vector<int>::iterator newEnd(remove(v.begin(), v.end(), 99));
```

this is what `v` looks like after the call:



Here I've used question marks to indicate the values of elements that have been conceptually removed from `v` but continue to exist.

It seems logical that if the "unremoved" elements are in `v` between `v.begin()` and `newEnd`, the "removed" elements must be between `newEnd` and `v.end()`. *This is not the case!* The "removed" values aren't necessarily in `v` any longer at all. `remove` doesn't change the order of the elements in a range so that all the "removed" ones are at the end, it arranges for all the "unremoved" values to be at the beginning. Though the Standard doesn't require it, the elements beyond the new logical end of the range typically *retain their old values*. After calling `remove`, `v looks` like this in every implementation I know:



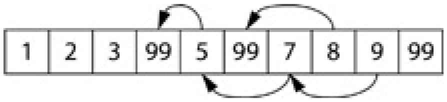
As you can see, two of the "99" values that used to exist in `v` are no longer there, while one "99" remains. In general, after calling `remove`, the values removed from the range may or may not continue to exist in the range. Most people find this surprising, but why? You asked `remove` to get rid of some values, so it did. You didn't ask it to put the removed values in a special place where you could get at them later, so it didn't. What's the problem? (If you don't want to lose any values, you should probably be calling `partition` or `stable_partition` instead of `remove`. `partition` and `stable_partition` are described in Item 31.)

`remove`'s behavior sounds spiteful, but it's simply a fallout of the way the algorithm operates. Internally, `remove` walks down the range, overwriting values that are to be "removed" with later values that are to be retained. The overwriting is accomplished by making assignments to the elements holding the values to be overwritten.

You can think of `remove` as performing a kind of compaction, where the values to be removed play the role of holes that are filled during compaction. For our vector `v`, it plays out as follows.

1. `remove` examines `v[0]`, sees that its value isn't supposed to be removed, and moves on to `v[1]`. It does the same for `v[1]` and `v[2]`.
2. It sees that `v[3]` should be removed, so it notes that `v[3]`'s value may be overwritten, and it moves on to `v[4]`. This is akin to noting that `v[3]` is a "hole" that needs to be filled.
3. It sees that `v[4]`'s value should be retained, so it assigns `v[4]` to `v[3]`, notes that `v[4]` may now be overwritten, and moves on to `v[5]`. Continuing the compaction analogy, it "fills" `v[3]` with `v[4]` and notes that `v[4]` is now a hole.
4. It finds that `v[5]` should be removed, so it ignores it and moves on to `v[6]`. It continues to remember that `v[4]` is a hole waiting to be filled.
5. It sees that `v[6]` is a value that should be kept, so it assigns `v[6]` to `v[4]`, remembers that `v[5]` is now the next hole to be filled, and moves on to `v[7]`.
6. In a manner analogous to the above, it examines `v[7]`, `v[8]` and `v[9]`. It assigns `v[7]` to `v[5]` and `v[8]` to `v[6]`, ignoring `v[9]`, because the value at `v[9]` is to be removed.
7. It returns an iterator indicating the next element to be overwritten, in this case the element at `v[7]`.

You can envision the values moving around in `v` as follows:



As Item 33 explains, the fact that `remove` overwrites some of the values it is removing has important repercussions when those values are pointers. For this Item, however, it’s enough to understand that `remove` doesn’t eliminate any elements from a container, because it can’t. Only container member functions can eliminate container elements, and that’s the whole point of this Item: You should follow `remove` by `erase` if you really want to remove something.

The elements you want to erase are easy to identify. They’re the elements of the original range that start at the “new logical end” of the range and continue until the real end of the range. To get rid of these elements, all you need to do is call the range form of erase (see Item 5) with these two iterators. Because `remove` itself conveniently returns the iterator for the new logical end of the range, the call is straightforward:

Click here to view code image

```
vector<int> v;                                       // as
before

...

v.erase(remove(v.begin(), v.end(), 99), v.end());    //
really remove all
                                                     //
elements with value 99

cout << v.size();                                    // now
returns 7
```

Passing `remove`’s return value as the first argument to the range form of `erase` is so common, it’s idiomatic. In fact, `remove` and `erase` are so closely allied, the two are merged in the `list` member function `remove`. This is the only function in the STL named `remove` that eliminates elements from a container:

Click here to view code image

```
list<int> li;                      // create a list

...                                // put some values into it
```

```
li.remove(99);                    // eliminate all elements
with value 99;
                                  // this really removes
elements, so li's
                                  // size may change
```

Frankly, calling this function `remove` is an inconsistency in the STL. The analogous function in the associative containers is called `erase`, and `list`’s `remove` should be called `erase`, too. It’s not, however, so we all have to get used to it. The world in which we frolic may not be the best of all possible worlds, but it is the one we’ve got. (On the plus side, Item 44 points out that, for lists, calling the `remove` member function is more efficient than applying the `erase-remove` idiom.)

Once you understand that `remove` can’t “really” remove things from a container, using it in conjunction with `erase` becomes second nature. The only other thing you need to bear in mind is that `remove` isn’t the only algorithm for which this is the case. There are two other “`remove`-like” algorithms: `remove_if` and `unique`.

The similarity between `remove` and `remove_if` is so straightforward, I won’t dwell on it, but `unique` also behaves like `remove`. It is asked to remove things (adjacent repeated values) from a range without having access to the container holding the range’s elements. As a result, you must also pair calls to `unique` with calls to `erase` if you really want to remove elements from a container. `unique` is also analogous to `remove` in its interaction with `list`. Just as `list::remove` really removes things (and does so more efficiently than the `erase-remove` idiom), `list::unique` really removes adjacent duplicates (also with greater efficiency than would `erase-unique`).

Item 33: Be wary of `remove`-like algorithms on containers of pointers

So you conjure up a bunch of dynamically allocated `Widget`s, each of which may be certified, and you store the resulting pointers in a `vector`:

Click here to view code image

```
class Widget {
public:
  ...
  bool isCertified() const;           // whether the Widget
is certified
  ...
};

vector<Widget*> v;                      // create a vector and
fill it with
...                                     // pointers to
dynamically
v.push_back(new Widget);                // allocated Widgets
...
```

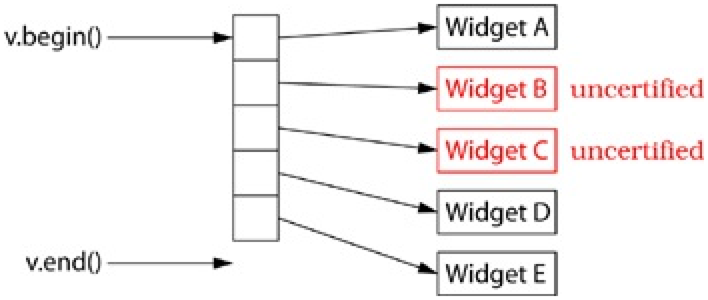
After working with `v` for a while, you decide to get rid of the uncertified Widgets, because you don't need them any longer. Bearing in mind Item 43's admonition to prefer algorithm calls to explicit loops and having read Item 32's discourse on the relationship between `remove` and `erase`, your thoughts naturally turn to the `erase-remove` idiom, though in this case it's `remove_if` you employ:

Click here to view code image

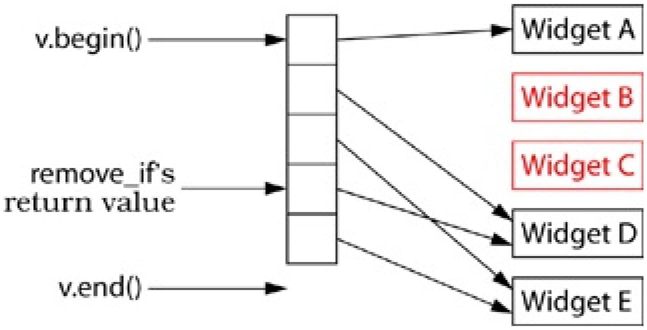
```
v.erase(remove_if(v.begin(), v.end(),                   //
erase ptrs to
                  not1(mem_fun(&Widget::isCertified))), //
uncertified
        v.end());                                       //
Widgets; see
                                                        //
Item 41 for
                                                        //
info on
                                                        //
mem_fun
```

Suddenly you begin to worry about the call to `erase`, because you dimly recall Item 7's discussion of how destroying a pointer in a container fails to delete what the pointer points to. This is a legitimate worry, but in this case, it comes too late. By the time `erase` is called, there's an excellent chance you have already leaked resources. Worry about `erase`, yes, but first, worry about `remove_if`.

Let's assume that prior to the `remove_if` call, `v` looks like this, where I've indicated the uncertified Widgets.


v.begin()
v.end()
Widget A
Widget B
uncertified
Widget C
uncertified
Widget D
Widget E

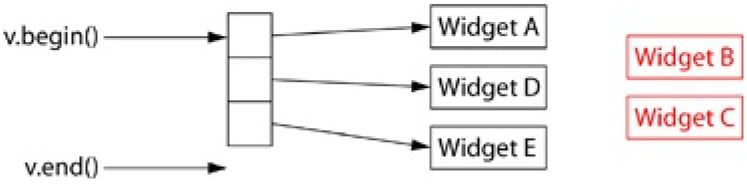
After the call to `remove_if`, `v` will typically look like this (including the iterator returned from `remove_if`):


v.begin()
remove_if's return value
v.end()
Widget A
Widget B
Widget C
Widget D
Widget E

If this transformation makes no sense, kindly turn to Item 32, because it explains exactly what happens when `remove`—or, in this case, `remove_if`—is called.

The reason for the resource leak should now be apparent. The “removed” pointers to Widgets B and C have been overwritten by later “unremoved” pointers in the vector. Nothing points to the two uncertified Widgets, they can never be deleted, and their memory and other resources are leaked.

Once both `remove_if` and `erase` have returned, the situations looks as follows:


v.begin()
v.end()
Widget A
Widget D
Widget E
Widget B
Widget C

This makes the resource leak especially obvious, and it should now be clear why you should try to avoid using `remove` and similar algorithms (i.e., `remove_if` and `unique`) on containers of dynamically allocated objects. In many cases, you’ll find that the `partition` algorithm (see Item 31) is a reasonable alternative.

If you can’t avoid using `remove` on such containers, one way to eliminate this problem is to delete the pointers and set them to null prior to applying the `erase-remove` idiom, then eliminate all the null pointers in the container:

[**Click here to view code image**]

```
void delAndNullifyUncertified(Widget*& pWidget)      // if
*pWidget is an
{                                                    //
uncertified Widget,
  if (!pWidget->isCertified()) {                     //
delete the pointer
      delete pWidget;                                // and
set it to null
      pWidget = 0;
  }

}

for_each(v.begin(), v.end(),                        // delete
and set to
         delAndNullifyUncertified);                 // null
all ptrs to
                                                    //
uncertified widgets

v.erase(remove(v.begin(), v.end(),                  //
eliminate null ptrs
               static_cast<Widget*>(0)),            // from
v; 0 must be
        v.end());                                   // cast
to a ptr so C++
                                                    //
correctly deduces
                                                    // the
type of
                                                    //
remove's 3rd param
```

Of course, this assumes that the vector doesn’t hold any null pointers you’d like to retain. If it does, you’ll probably have to write your own loop that erases pointers as you go. Erasing elements from a container as you traverse that container has some subtle aspects to it, so be sure to read Item 9 before considering that approach.

If you’re willing to replace the container of pointers with a container of *smart* pointers that perform reference counting, the `remove`-related difficulties wash away, and you can use the `erase-remove` idiom directly:

Click here to view code image

```
template<typename T>                    // RCSP = "Reference
Counting
class RCSP { ... };                     //         Smart
Pointer"

typedef RCSP<Widget> RCSPW;             // RCSPW = "RCSP to
Widget"

vector<RCSPW> v;                        // create a vector
and fill it with
...                                     // smart pointers to
dynamically
v.push_back(RCSPW(new Widget));         // allocated Widgets
...

v.erase(remove_if(v.begin(), v.end(),                   //
erase the ptrs
                  not1(mem_fun(&Widget::isCertified))), // to
uncertified
         v.end());                                      //
Widgets; no
                                                        //
resources are
                                                        //
leaked
```

For this to work, it must be possible to implicitly convert your smart pointer type (e.g., `RCSP<Widget>`) to the corresponding built-in pointer type (e.g., `Widget*`). That’s because the container holds smart pointers, but the member function being called (e.g., `Widget::isCertified`) insists on built-in pointers. If no implicit conversion exists, your compilers will squawk.

If you don’t happen to have a reference counting smart pointer template in your programming toolbox, you owe it to yourself to check out the `shared_ptr` template in the Boost library. For an introduction to Boost, take a look at Item 50.

Regardless of how you choose to deal with containers of dynamically allocated pointers, be it by reference counting smart pointers, manual deletion and nullification of pointers prior to invoking a `remove`-like algorithm, or some technique of your own invention, the guidance of this Item remains the same: Be wary of `remove`-like algorithms on containers of pointers. Failure to heed this advice is just *asking* for resource leaks.

Item 34: Note which algorithms expect sorted ranges

Not all algorithms are applicable to all ranges. For example, `remove` (see Items 32 and 33) requires forward iterators and the ability to make assignments through those iterators. As a result, it can't be applied to ranges demarcated by input iterators, nor to `map`s or `multimap`s, nor to some implementations of `set` and `multiset` (see Item 22). Similarly, many of the sorting algorithms (see Item 31) demand random access iterators, so it's not possible to invoke these algorithms on the elements of a `list`.

If you violate these kinds of rules, your code won't compile, an event likely to be heralded by lengthy and incomprehensible error messages (see Item 49). Other algorithm preconditions, however, are more subtle. Among these, perhaps the most common is that some algorithms require ranges of *sorted* values. It's important that you adhere to this requirement whenever it applies, because violating it leads not to compiler diagnostics, but to undefined runtime behavior.

A few algorithms can work with sorted or unsorted ranges, but they are most useful when they operate on sorted ranges. You should understand how these algorithms work, because that will explain why sorted ranges suit them best.

Some of you, I know, are into brute-force memorization, so here's a list of the algorithms that require the data on which they operate to be sorted:

```
binary_search            lower_bound
upper_bound              equal_range

set_union                set_intersection
set_difference           set_symmetric_difference

merge                    inplace_merge

includes
```

In addition, the following algorithms are typically used with sorted ranges, though they don't require them:

`unique` `unique_copy`

We’ll see shortly that the definition of “sorted” has an important constraint, but first, let me try to make sense of this collection of algorithms. It’s easier to remember which algorithms work with sorted ranges if you understand why such ranges are needed.

The search algorithms `binary_search`, `lower_bound`, `upper_bound`, and `equal_range` (see Item 45) require sorted ranges, because they look for values using binary search. Like the C library’s `bsearch`, these algorithms promise logarithmic-time lookups, but in exchange, you must give them values that have already been put into order.

Actually, it’s not quite true that these algorithms promise logarithmic-time lookup. They guarantee such performance only when they are passed random access iterators. If they’re given less powerful iterators (such as bidirectional iterators), they still perform only a logarithmic number of comparisons, but they run in linear time. That’s because, lacking the ability to perform “iterator arithmetic,” they need linear time to move from place to place in the range being searched.

The quartet of algorithms `set_union`, `set_intersection`, `set_difference`, and `set_symmetric_difference` offer linear-time performance of the set-theoretical operations their names suggest. Why do they demand sorted ranges? Because without them, they couldn’t do their work in linear time. If you’re beginning to detect a trend suggesting that algorithms requiring sorted ranges do so in order to offer better performance than they’d be able to guarantee for ranges that might not be sorted, you’re right. Stay tuned. The trend will continue.

`merge` and `inplace_merge` perform what is in effect a single pass of the mergesort algorithm: they read two sorted ranges and produce a new sorted range containing all the elements from both source ranges. They run in linear time, something they couldn’t do if they didn’t know that the source ranges were already sorted.

The final algorithm that requires sorted source ranges is `includes`. It’s used to determine whether all the objects in one range are also in another range. Because `includes` may assume that both its ranges are sorted, it promises linear-time performance. Without that guarantee, it would generally run slower.

Unlike the algorithms we’ve just discussed, `unique` and `unique_copy` offer well-defined behavior even on unsorted ranges. But look at how the Standard describes `unique`'s behavior (the italics are mine):

> Eliminates all but the first element from every *consecutive* group of equal elements.

In other words, if you want `unique` to eliminate all duplicates from a range (i.e., to make all values in the range "unique"), you must first make sure that all duplicate values are next to one another. And guess what? That's one of the things sorting does. In practice, `unique` is usually employed to eliminate all duplicate values from a range, so you'll almost always want to make sure that the range you pass `unique` (or `unique_copy`) is sorted. (Unix developers will recognize a striking similarity between STL's `unique` and Unix's `uniq`, a similarity I suspect is anything but coincidental.)

Incidentally, `unique` eliminates elements from a range the same way `remove` does, which is to say that it only "sort of" eliminates them. If you aren't sure what this means, please turn to Items 32 and 33 immediately. It is not possible to overemphasize the importance of understanding what `remove` and `remove`-like algorithms (including `unique`) do. Having a basic comprehension is not sufficient. If you don't *know* what they do, you will get into trouble.

Which brings me to the fine print regarding what it means for a range to be sorted. Because the STL allows you to specify comparison functions to be used during sorting, different ranges may be sorted in different ways. Given two ranges of `int`s, for example, one might be sorted the default way (i.e., in ascending order) while the other is sorted using `greater<int>`, hence in descending order. Given two ranges of `Widget`s, one might be sorted by price and another might be sorted by age. With so many different ways to sort things, it's critical that you give the STL consistent sorting-related information to work with. If you pass a sorted range to an algorithm that also takes a comparison function, be sure that the comparison function you pass behaves the same as the one you used to sort the range.

Here's an example of what you do *not* want to do:

Click here to view code image

```
vector<int> v;                              // create a
vector, put some
...                                         // data into
it, sort it into
```

```
sort(v.begin(), v.end(), greater<int>());     // descending
order

...                                           // work with
the vector
                                              // (without
changing it)

bool a5Exists =                               // search for a
5 in the vector,
  binary_search(v.begin(), v.end(), 5);       // assuming
it's sorted in
                                              // ascending
order!
```

By default, `binary_search` assumes that the range it's searching is sorted by "<" (i.e., the values are in ascending order), but in this example, the vector is sorted in *descending* order. You should not be surprised to learn that you get undefined results when you invoke `binary_search` (or `lower_bound`, etc.) on a range of values that is sorted in a different order from what the algorithm expects.

To get the code to behave correctly, you must tell `binary_search` to use the same comparison function that `sort` did:

Click here to view code image

```
bool a5Exists =                                           //
search for a 5
  binary_search(v.begin(), v.end(), 5, greater<int>());   //
using greater as
                                                          //
the comparison
                                                          //
function
```

All the algorithms that require sorted ranges (i.e., all the algorithms in this Item except `unique` and `unique_copy`) determine whether two values are "the same" by using equivalence, just like the standard associative containers (which are themselves sorted). In contrast, the default way in which `unique` and `unique_copy` determine whether two objects are "the same" is by using equality, though you can override this default by passing these algorithms a predicate defining an alternative definition of "the same." For a detailed discussion of the difference between equivalence and equality, consult Item

19.

The eleven algorithms that require sorted ranges do so in order to offer greater efficiency than would otherwise be possible. As long as you remember to pass them only sorted ranges, and as long as you make sure that the comparison function used by the algorithms is consistent with the one used to do the sorting, you'll revel in trouble-free search, set, and merge operations, plus you'll find that `unique` and `unique_copy` eliminate *all* duplicate values, as you almost certainly want them to.

Item 35: Implement simple case-insensitive string comparisons via `mismatch` or `lexicographical_compare`

One of the most frequently asked questions by STL newbies is "How do I use the STL to perform case-insensitive string comparisons?" This is a deceptively simple question. Case-insensitive string comparisons are either really easy or really hard, depending on how general you want to be. If you're willing to ignore internationalization issues and restrict your concern to the kinds of strings `strcmp` is designed for, the task is easy. If you want to be able to handle strings of characters in languages where `strcmp` wouldn't apply (i.e., strings holding text in just about any language except English) or where programs use a locale other than the default, the task is very hard.

In this Item, I'll tackle the easy version of the problem, because that suffices to demonstrate how the STL can be brought to bear. (A harder version of the problem involves no more of the STL. Rather, it involves locale-dependent issues you can read about in Appendix A.) To make the easy problem somewhat more challenging, I'll tackle it twice. Programmers desiring case-insensitive string comparisons often need two different calling interfaces, one similar to `strcmp` (which returns a negative number, zero, or a positive number), the other akin to `operator<` (which returns `true` or `false`). I'll therefore show how to implement both calling interfaces using STL algorithms.

First, however, we need a way to determine whether two characters are the same, except for their case. When internationalization issues are taken into account, this is a complicated problem. The following character-comparing function is a simplistic solution, but it's akin to the `strcmp` approach to string comparison, and since in this Item I consider only strings where a `strcmp`-like approach is appropriate, internationalization issues don’t count, and this function will do:

Click here to view code image

```
int ciCharCompare(char c1, char c2)    // case-insensitively
compare chars
{                                      // c1 and c2,
returning -1 if c1 < c2,
                                       // 0 if c1==c2, and 1
if c1 > c2

  int lc1 = tolower(static_cast<unsigned char>(c1));      //
see below for
  int lc2 = tolower(static_cast<unsigned char>(c2));      //
info on these
                                                          //
statements

  if (lc1 < lc2) return -1;

  if (lc1 > lc2) return 1;

  return 0;
}
```

This function follows the lead of `strcmp` in returning a negative number, zero, or a positive number, depending on the relationship between `c1` and `c2`. Unlike `strcmp`, `ciCharCompare` converts both parameters to lower case before performing the comparison. That’s what makes it a case-insensitive character comparison.

Like many functions in `<cctype>` (and hence `<ctype.h>`), `tolower`’s parameter and return value is of type `int`, but unless that int is `EOF`, its value must be representable as an `unsigned char`. In both C and C++, `char` may or may not be signed (it’s up to the implementation), and when `char` is signed, the only way to ensure that its value is representable as an `unsigned char` is to cast it to one before calling `tolower`. That explains the casts in the code above. (On implementations where `char` is already unsigned, the casts are no-ops.) It also explains the use of `int` instead of `char` to store `tolower`’s return value.

Given `ciCharCompare`, it’s easy to write the first of our two case-insensitive string comparison functions, the one offering a `strcmp`-like interface. This function, `ciStringCompare`, returns a negative number, zero, or a positive number, depending on the relationship between the strings being compared. It’s built around the `mismatch` algorithm, because `mismatch` identifies the first position in two ranges where the corresponding values are not the same.

Before we can call `mismatch`, we have to satisfy its preconditions. In particular, we have to make sure that if one string is shorter than the other, the shorter string is the first range passed. We’ll therefore farm the real work out to a function called `ciStringCompareImpl` and have `ciStringCompare` simply make sure the arguments are passed in the correct order, adjusting the return value if the arguments have to be swapped:

Click here to view code image

```
int ciStringCompareImpl(const string& s1,                 //
see below for
                        const string& s2);                //
implementation

int ciStringCompare(const string& s1, const string& s2)
{
  if (s1.size() <= s2.size()) return ciStringCompareImpl(s1,
s2);
  else return -ciStringCompareImpl(s2, s1);
}
```

In `ciStringCompareImpl`, the heavy lifting is performed by `mismatch`. It returns a pair of iterators indicating the locations in the ranges where corresponding characters first fail to match:

Click here to view code image

```
int ciStringCompareImpl(const string& s1, const string& s2)
{
  typedef pair<string::const_iterator,            // PSCI =
"pair of
               string::const_iterator> PSCI;      //
string::const_iterator"

  PSCI p = mismatch(                              // see
below for an
            s1.begin(), s1.end(),                 //
explanation of why
            s2.begin(),                           // we need
not2; see
            not2(ptr_fun(ciCharCompare)));        // Item 41
for why we
```

```
                                                  // need
ptr_fun

  if (p.first == s1.end()) {                      // if true,
either s1 and
     if (p.second == s2.end()) return 0;          // s2 are
equal or
     else return -1;                              // s1 is
shorter than s2
  }

  return ciCharCompare(*p.first, *p.second);      // the
relationship of the
}                                                 // strings
is the same as
                                                  // that of
the
                                                  //
mismatched chars
```

With any luck, the comments make pretty clear what is going on. Fundamentally, once you know the first place where the strings differ, it’s easy to determine which string, if either, precedes the other. The only thing that may seem odd is the predicate passed to `mismatch`, which is `not2(ptr_fun(ciCharCompare))`. This predicate is responsible for returning `true` when the characters match, because `mismatch` will stop when the predicate returns `false`. We can’t use `ciCharCompare` for this purpose, because it returns -1, 1, or 0, and *it returns 0 when the characters match*, just like `strcmp`. If we passed `ciCharCompare` as the predicate to `mismatch`, `C++` would convert `ciCharCompare`’s return type to `bool`, and of course the `bool` equivalent of zero is `false`, precisely the opposite of what we want! Similarly, when `ciCharCompare` returned 1 or –1, that would be interpreted as `true`, because, as in C, all nonzero integral values are considered `true`. Again, this would be the opposite of what we want. To fix this semantic inversion, we throw a `not2` and a `ptr_fun` in front of `ciCharCompare`, and we all live happily ever after.

Our second approach to `ciStringCompare` yields a conventional STL predicate; such a function could be used as a comparison function in associative containers. The implementation is short and sweet, because all we have to do is modify `ciCharCompare` to give us a character-comparison function with a predicate interface, then turn the job of performing a string comparison over to the algorithm with the second-longest name in the STL, `lexicographical_compare`:

Click here to view code image

```
bool ciCharLess(char c1, char c2)               // return
whether c1
{                                               // precedes
c2 in a case-
  return                                        //
insensitive comparison;
    tolower(static_cast<unsigned char>(c1)) <   // Item 46
explains why a
    tolower(static_cast<unsigned char>(c2));    // function
object might
}                                               // be
preferable to this
                                                // function

bool ciStringCompare(const string& s1, const string& s2)
{
  return lexicographical_compare(s1.begin(), s1.end(),  //
see below for
                                 s2.begin(), s2.end(),  // a
discussion of
                                 ciCharLess);           //
this algorithm
}                                                       //
call
```

No, I won't keep you in suspense any longer. The longest algorithm name is `set_symmetric_difference`.

If you're familiar with the behavior of `lexicographical_compare`, the code above is as clear as clear can be. If you're not, it's probably about as clear as concrete. Fortunately, it's not hard to replace the concrete with glass.

`lexicographical_compare` is a generalized version of `strcmp`. Where `strcmp` works only with character arrays, however, `lexicographical_compare` works with ranges of values of any type. Also, while `strcmp` always compares two characters to see if their relationship is equal, less than, or greater than one another, `lexicographical_compare` may be passed an arbitrary predicate that determines whether two values satisfy a user-defined criterion.

In the call above, `lexicographical_compare` is asked to find the first position where `s1` and `s2` differ, based on the results of calls to `ciCharLess`. If, using the characters at that position, `ciCharLess` returns `true`, so does `lexicographical_compare`: if, at the first position where the characters differ, the character from the first string precedes the corresponding character from the second string, the first string precedes the second one. Like `strcmp`, `lexicographical_compare` considers two ranges of equal values to be equal, hence it returns `false` for such ranges: the first range does *not* precede the second. Also like `strcmp`, if the first range ends before a difference in corresponding values is found, `lexicographical_compare` returns `true`: a prefix precedes any range for which it is a prefix.

Enough about `mismatch` and `lexicographical_compare`. Though I focus on portability in this book, I would be remiss if I failed to mention that case-insensitive string comparison functions are widely available as nonstandard extensions to the standard C library. They typically have names like `stricmp` or `strcmpi`, and they typically offer no more support for internationalization than the functions we've developed in this Item. If you're willing to sacrifice some portability, you know that your strings never contain embedded nulls, and you don't care about internationalization, you may find that the easiest way to implement a case-insensitive string comparison doesn't use the STL at all. Instead, it converts both strings to `const char*` pointers (see Item 16) and then uses `stricmp` or `strcmpi` on the pointers:

Click here to view code image

```
int ciStringCompare(const string& s1, const string& s2)
{
  return stricmp(s1.c_str(), s2.c_str());         // the
function name on
}                                                 // your
system might
                                                  // not be
stricmp
```

Some may call this a hack, but `stricmp/strcmpi`, being optimized to do exactly one thing, typically run *much* faster on long strings than do the general-purpose algorithms `mismatch` and `lexicographical_compare`. If that's an important consideration for you, you may not care that you're trading standard STL algorithms for nonstandard C functions. Sometimes the most effective way to use the STL is to realize that other approaches are superior.

Item 36: Understand the proper implementation of `copy_if`

One of the more interesting aspects of the STL is that although there are 11 algorithms with “copy” in their names,

Click here to view code image

```
copy                    copy_backward
replace_copy            reverse_copy
replace_copy_if         unique_copy
remove_copy             rotate_copy
remove_copy_if          partial_sort_copy
uninitialized_copy
```

none of them is `copy_if`. That means you can `replace_copy_if`, you can `remove_copy_if`, you can both `copy_backward` and `reverse_copy`, but if you simply want to copy the elements of a range that satisfy a predicate, you’re on your own.

For example, suppose you have a function to determine whether a `Widget` is defective:

```
bool isDefective(const Widget& w);
```

and you’d like to write all the defective Widgets in a vector to `cerr`. If `copy_if` existed, you could simply do this:

Click here to view code image

```
vector<Widget> widgets;

...

copy_if(widgets.begin(), widgets.end(),          // this
won't compile;
        ostream_iterator<Widget>(cerr, "\n"),    // there is
no copy_if
        isDefective);                            // in the
STL
```

Ironically, `copy_if` was part of the original Hewlett Packard STL that formed the basis for the STL that is now part of the standard C++ library. In one of those quirks that occasionally makes history interesting, during the process of winnowing the HP STL into something of a size manageable for standardization, `copy_if` was one of the things that got left on the cutting room floor.

In *The C++ Programming Language* [7], Stroustrup remarks that it’s trivial to write `copy_if`, and he’s right, but that doesn’t mean that it’s necessarily easy to come up with the correct trivia. For example, here’s a reasonable-looking `copy_if` that many people (including me) have been known to come up with:

Click here to view code image

```
template< typename InputIterator,               // a not-
quite-right
          typename OutputIterator,              //
implementation of
          typename Predicate>                   // copy_if
OutputIterator copy_if(InputIterator begin,
                       InputIterator end,
                       OutputIterator destBegin,
                       Predicate p)
{
  return remove_copy_if(begin, end, destBegin, not1(p));
}
```

This approach is based on the observation that although the STL doesn’t let you say “copy everything where this predicate is true,” it does let you say “copy everything except where this predicate is *not* true.” To implement `copy_if`, then, it seems that all we need to do is throw a `not1` in front of the predicate we’d like to pass to `copy_if`, then pass the resulting predicate to `remove_copy_if`. The result is the code above.

If the above reasoning were valid, we could write out our defective `Widget`s this way:

Click here to view code image

```
copy_if(widgets.begin(), widgets.end(),         // well-
intentioned code
        ostream_iterator<Widget>(cerr, "\n"),   // that will
not compile
        isDefective);
```

Your STL platforms will take a jaundiced view of this code, because it tries to apply `not1` to `isDefective`. (The application takes place inside `copy_if`). As Item 41 tries to make clear, `not1` can’t be applied directly to a function pointer; the function pointer must first be passed through `ptr_fun`. To call this implementation of `copy_if`, you must pass not just a function object, but an *adaptable* function object. That’s easy enough to do, but clients of a would-be STL algorithm shouldn’t have to. Standard STL algorithms never require that their functors be adaptable, and neither should `copy_if`. The above implementation is decent, but it’s not decent enough.

Here's the correct trivial implementation of `copy_if`:

Click here to view code image

```
template< typename InputIterator,                    // a correct
          typename OutputIterator,                   //
implementation of
          typename Predicate>                        // copy_if
OutputIterator copy_if(InputIterator begin,
                       InputIterator end,
                       OutputIterator destBegin,
                       Predicate p)
{
  while (begin != end) {
   if (p(*begin)) *destBegin++ = *begin;
   ++begin;
  }

  return destBegin;
}
```

Given how useful `copy_if` is, plus the fact that new STL programmers tend to expect it to exist anyway, there's a good case to be made for putting `copy_if`—the correct one!—into your local STL-related utility library and using it whenever it's appropriate.

Item 37: Use `accumulate` or `for_each` to summarize ranges

Sometimes you need to boil an entire range down to a single number, or, more generally, a single object. For commonly needed information, special-purpose algorithms exist to do the jobs. `count` tells you how many elements with a particular value are in a range, for example, while `count_if` tells you how many elements satisfy a predicate. The minimum and maximum values in a range are available via `min_element` and `max_element`.

At times, however, you need to summarize a range in some custom manner, and in those cases, you need something more flexible than `count`, `count_if`, `min_element`, or `max_element`. For example, you might want the sum of the lengths of the strings in a container. You might want the product of a range of numbers. You might want the average coordinates of a range of points. In each of these cases, you need to *summarize* a range, but you need to be able to define the summary you want. Not a problem. The STL has the algorithm for you. It's called `accumulate`. You might not be familiar with `accumulate`, because, unlike most algorithms, it doesn’t live in `<algorithm>`. Instead, it’s located with three other “numeric algorithms” in `<numeric>`. (The three others are `inner_product`, `adjacent_difference`, and `partial_sum`.)

Like many algorithms, `accumulate` exists in two forms. The form taking a pair of iterators and an initial value returns the initial value plus the sum of the values in the range demarcated by the iterators:

Click here to view code image

```
list<double> ld;                                    // create
a list and put
...                                                 // some
doubles into it

double sum = accumulate(ld.begin(), ld.end(), 0.0); //
calculate their sum,
                                                    //
starting at 0.0
```

In this example, note that the initial value is specified as `0.0`, not simply `0`. That’s important. The type of `0.0` is `double`, so `accumulate` internally uses a variable of type `double` to store the sum it’s computing. Had the call been written like this,

Click here to view code image

```
double sum = accumulate(ld.begin(), ld.end(), 0);   //
calculate their sum,
                                                    //
starting at 0; this
                                                    // is not
correct!
```

the initial value would be the `int 0`, so `accumulate` would internally use an `int` to store the value it was computing. That `int` would ultimately become `accumulate`’s return value, and it would be used to initialize the variable `sum`. The code would compile and run, but `sum`’s value would be incorrect. Instead of holding the true sum of a list of `double`s, it would hold the result of adding all the `double`s together, but converting the result to an `int` after each addition.

`accumulate` requires only input iterators, so you can use it even with `istream_iterator`s and `istreambuf_iterator`s (see Item 29):

Click here to view code image

```
cout << "The sum of the ints on the standard input is "  //
```

```
print the sum of
     << accumulate(istream_iterator<int>(cin),           //
the ints in cin
                   istream_iterator<int>(),
                   0);
```

It is this default behavior of `accumulate` that causes it to be labeled a numeric algorithm. But when `accumulate` is used in its alternate form, one taking an initial summary value and an arbitrary summarization function, it becomes much more general.

As an example, consider how to use `accumulate` to calculate the sum of the lengths of the strings in a container. To compute the sum, `accumulate` needs to know two things. First, as above, it must know the starting sum. In our case, it is zero. Second, it must know how to update this sum each time a new string is seen. To do that, we write a function that takes the sum so far and the new string and returns the updated sum:

Click here to view code image

```
string::size_type                              // see below
for info
stringLengthSum(string::size_type sumSoFar,    // on
string::size_type
                const string& s)
{
  return sumSoFar + s.size();
}
```

The body of this function reveals that what is going on is trivial, but you may find yourself bogged down in the appearances of `string::size_type`. Don’t let that happen. Every standard STL container has a typedef called `size_type` that is the container’s type for counting things. This is the type returned by the container’s `size` function, for example. For all the standard containers, `size_type` must be `size_t`, but, in theory, nonstandard STL-compatible containers might use a different type for `size_type` (though I have a hard time imagining why they’d want to). For standard containers, you can think of *`Container`*`::size_type` as a fancy way of writing `size_t`.

`stringLengthSum` is representative of the summarization functions `accumulate` works with. It takes a summary value for the range so far as well as the next element of the range, and it returns the new summary value. In general, that means the function will take parameters of different types. That’s what it does here. The summary so far (the sum of the lengths of the strings already seen) is of type `string::size_type`, while the type of the elements being examined is `string`. As is typically the case, the return type here is the same as that of the function’s first parameter, because it’s the updated summary value (the one taking the latest element into account).

We can use `stringLengthSum` with `accumulate` like this:

Click here to view code image

```
set<string> ss;                          // create container
of strings,
...                                      // and populate it

string::size_type lengthSum =            // set lengthSum to
the result
  accumulate(ss.begin(), ss.end(),       // of calling
stringLengthSum on
    static_cast<string::size_type>(0),   // each element in
ss, using 0
    stringLengthSum);                    // as the initial
summary value
```

Nifty, huh? Calculating the product of a range of numbers is even easier, because we don’t have to write our own summation function. We can use the standard `multiplies` functor class:

Click here to view code image

```
vector<float> vf;                         // create
container of floats
...                                       // and populate it

float product =                           // set product to
the result of
  accumulate(vf.begin(), vf.end(),        // calling
multiplies<float> on
              1.0f, multiplies<float>()); // each element in
vf, using 1.0f
                                          // as the initial
summary value
```

The only tricky thing here is remembering to use one (as a `float`, not as an `int`!) as the initial summary value instead of zero. If we used zero as the starting value, the result would always be zero, because zero times anything is zero, right?

Our final example is a bit more ambitious. It involves finding the average of a range of points, where a point looks like this:

Click here to view code image

```
struct Point {
  Point(double initX, double initY): x(initX), y(initY) {}
  double x, y;
};
```

The summation function will be an object of a functor class called `PointAverage`, but before we look at `PointAverage`, let's look at its use in the call to `accumulate`:

Click here to view code image

```
list<Point> lp;
...

Point avg =                                              // average the
points in lp
  accumulate(lp.begin(), lp.end(),
             Point(0, 0), PointAverage());
```

Simple and straightforward, the way we like it. In this case, the initial summary value is a `Point` object located at the origin, and all we need to remember is not to take that point into account when computing the average of the range.

`PointAverage` works by keeping track of the number of points it has seen, as well as the sum of their `x` and `y` components. Each time it is called, it updates these values and returns the average coordinates of the points so far examined. Because it is called exactly once for each point in the range, it divides the `x` and `y` sums by the number of points in the range; the initial point value passed to `accumulate` is ignored, as it should be:

Click here to view code image

```
class PointAverage:
  public binary_function<Point, Point, Point>
{              // see Item 40
public:
  PointAverage(): numPoints(0), xSum(0), ySum(0) {}

  const Point operator()(const Point& avgSoFar, const Point&
p)
  {
    ++numPoints;
    xSum += p.x;
    ySum += p.y;

    return Point(xSum/numPoints, ySum/numPoints);
  }

private:
  size_t numPoints;
  double xSum;
  double ySum;
};
```

This works fine, and it is only because I sometimes associate with an inordinately demented group of people (many of them on the Standardization Committee) that I can envision STL implementations where it could fail. Nevertheless, `PointAverage` runs afoul of paragraph 2 of section 26.4.1 of the Standard, which, as I'm sure you recall, forbids side effects in the function passed to `accumulate`. Modification of the member variables `numPoints`, `xSum`, and `ySum` constitutes a side effect, so, technically speaking, the code I've just shown you yields undefined results. In practice, it's hard to imagine it not working, but I'm surrounded by menacing language lawyers as I write this, so I've no choice but to spell out the fine print on this matter.

That's okay, because it gives me a chance to mention `for_each`, another algorithm that can be used to summarize ranges and one that isn't constrained by the restrictions imposed on `accumulate`. Like `accumulate`, `for_each` takes a range and a function (typically a function object) to invoke on each element of the range, but the function passed to `for_each` receives only a single argument (the current range element), and `for_each` returns its function when it's done. (Actually, it returns a *copy* of its function—see Item 38.) Significantly, the function passed to (and later returned from) `for_each` *may* have side effects.

Ignoring the side effects issue, `for_each` differs from `accumulate` in two primary ways. First, the name `accumulate` suggests an algorithm that produces a summary of a range. `for_each` sounds like you simply want to do something to every element of a range, and, of course, that is the algorithm's primary application. Using `for_each` to summarize a range is legitimate, but it's not as clear as `accumulate`.

Second, `accumulate` returns the summary we want directly, while `for_each` returns a function object, and we must extract the summary information we want from this object. In C++ terms, that means we must add a member function to the functor class to let us retrieve the summary information we're after.

Here's the last example again, this time using `for_each` instead of `accumulate`:

Click here to view code image

```
struct Point { ... };                               // as
before

class PointAverage:
  public unary_function<Point, void> {              // see
Item 40
public:
  PointAverage(): numPoints(0), xSum(0), ySum(0) {}

  void operator()(const Point& p)
  {
    ++numPoints;
    xSum += p.x;
    ySum += p.y;
  }

  Point result() const
  {
    return Point(xSum/numPoints, ySum/numPoints);
  }

private:
  size_t numPoints;
  double xSum;
  double ySum;
};

list<Point> lp;
...

Point avg = for_each(lp.begin(), lp.end(),
PointAverage()).result();
```

Personally, I prefer `accumulate` for summarizing, because I think it most clearly expresses what is going on, but `for_each` works, too, and the issue of side effects doesn't dog `for_each` as it does `accumulate`. Both algorithms can be used to summarize ranges. Use the one that suits you best.

You may be wondering why `for_each`’s function parameter is allowed to have side effects while `accumulate`’s is not. This is a probing question, one that strikes at the heart of the STL. Alas, gentle reader, there are some mysteries meant to remain beyond our ken. Why the difference between `accumulate` and `for_each`? I’ve yet to hear a convincing explanation.

Chapter 6. Functors, Functor Classes, Functions, etc

Like it or not, functions and function-like objects—*functors*—pervade the STL. Associative containers use them to keep their elements in order, some algorithms (e.g., `find_if`) use them to control their behavior, others (e.g., `for_each` and `transform`) are meaningless without them, and adapters like `not1` and `bind2nd` actively produce them.

Yes, everywhere you look in the STL, you see functors and functor classes. Including in your source code. It's not possible to make effective use of the STL without knowing how to write well-behaved functors. That being the case, most of this chapter is devoted to explaining how to make your functors behave the way the STL expects them to. One Item, however, is devoted to a different topic, one sure to appeal to those who've wondered about the need to litter their code with `ptr_fun`, `mem_fun`, and `mem_fun_ref`. Start with that Item (Item 41), if you like, but please don't stop there. Once you understand those functions, you'll need the information in the remaining Items to ensure that your functors work properly with them, as well as with the rest of the STL.

Item 38: Design functor classes for pass-by-value

Neither C nor C++ allows you to truly pass functions as parameters to other functions. Instead, you must pass *pointers* to functions. For example, here's a declaration for the standard library function `qsort`:

Click here to view code image

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*cmpfcn)(const void*, const void*));
```

Item 46 explains why the `sort` algorithm is typically a better choice than the `qsort` function, but that's not at issue here. What is at issue is the declaration for `qsort`'s parameter `cmpfcn`. Once you've squinted past all the asterisks, it becomes clear that the argument passed as `cmpfcn`, which is a pointer to a function, is copied (i.e., passed by value) from the call site to `qsort`. This is representative of the rule followed by the standard libraries for both C and C++, namely, that function pointers are passed by value.

STL function objects are modeled after function pointers, so the convention in the STL is that function objects, too, are passed by value (i.e., copied) when passed to and from functions. This is perhaps best demonstrated by the Standard's declaration of `for_each`, an algorithm which both takes and returns a function object by value:

Click here to view code image

```
template<class InputIterator,
         class Function>
Function                                 // note return-by-value
for_each(InputIterator first,
         InputIterator last,
         Function f);                    // note pass-by-value
```

In truth, the pass-by-value case is not quite this iron-clad, because `for_each`'s caller could explicitly specify the parameter types at the point of the call. For example, the following would cause `for_each` to pass and return its functor by reference:

Click here to view code image

```
class DoSomething:
  public unary_function<int, void> {          // Item 40
explains base class
public:
  void operator()(int x) { ... }

  ...
};

typedef deque<int>::iterator DequeIntIter;  // convenience
typedef

deque<int> di;

...

DoSomething d;                                // create a
function object

...

for_each<DequeIntIter,                        // call for_each
with type
         DoSomething&>(di.begin(),            // parameters of
DequeIntIter
                       di.end(),            // and
DoSomething&;
                       d);                  // this forces d
to be
                                            // passed and
returned
                                            // by reference
```

Users of the STL almost never do this kind of thing, however, and some implementations of some STL algorithms won’t even compile if function objects are passed by reference. For the remainder of this Item, I’m going to pretend that function objects are always passed by value. In practice, that’s virtually always true.

Because function objects are passed and returned by value, the onus is on you to make sure that your function objects behave well when passed that way (i.e., copied). This implies two things. First, your function objects need to be small. Otherwise they will be too expensive to copy. Second, your function objects must be *monomorphic* (i.e., not polymorphic)—they must not use virtual functions. That’s because derived class objects passed by value into parameters of base class type suffer from the slicing problem: during the copy, their derived parts are removed. (For another example of how the slicing problem affects your use of the STL, see Item 3.)

Efficiency is important, of course, and so is avoiding the slicing problem, but not all functors are small, and not all are monomorphic, either. One of the advantages of function objects over real functions is that functors can contain as much state as you need. Some function objects are naturally hefty, and it’s important to be able to pass such functors to STL algorithms with the same ease as we pass their anorexic counterparts.

The prohibition on polymorphic functors is equally unrealistic. C++ supports inheritance hierarchies and dynamic binding, and these features are as useful when designing functor classes as anyplace else. Functor classes without inheritance would be like, well, like C++ without the “++”. Surely there’s a way to let function objects be big and/or polymorphic, yet still allow them to mesh with the pass-functors-by-value convention that pervades the STL.

There is. Take the data and/or the polymorphism you’d like to put in your functor class, and move it into a different class. Then give your functor class a pointer to this new class. For example, if you’d like to create a polymorphic functor class containing lots of data,

Click here to view code image

```
template<typename T>
class BPFC:                                         // BPFC =
"Big Polymorphic
  public                                            //          Fu
Class"
    unary_function<T, void> {                       // Item 40
explains this
                                                    // base class

private:
  Widget w;                                         // this class
has lots of data,
  int x;                                            // so it
would be inefficient
  ...                                               // to pass it
by value

public:
  virtual void operator()(const T& val) const;     // this is a
virtual function,
  ...                                               // so slicing
would be bad
};
```

create a small, monomorphic class that contains a pointer to an implementation class, and put all the data and virtual functions in the implementation class:

Click here to view code image

```
template<typename T>                                // new
implementation class
class BPFCImpl:                                     // for
modified BPFC
  public unary_function<T, void> {
private:
  Widget w;                                         // all the
data that used to
  int x;                                            // be in BPFC
are now here
  ...

  virtual ~BPFCImpl();                              //
polymorphic classes need
```

```
                                                // virtual
destructors
  virtual void operator()(const T& val) const;

friend class BPFC<T>;                           // let BPFC
access the data
};

template<typename T>
class BPFC:                                     // small,
monomorphic
  public unary_function<T, void> {              // version of
BPFC
private:
  BPFCImpl<T> *pImpl;                           // this is
BPFC's only data

public:
  void operator()(const T& val) const           // this is
now nonvirtual;
  {                                             // it
forwards to BPFCImpl
    pImpl->operator()(val);
  }
  ...
};
```

The implementation of `BPFC::operator()` exemplifies how all `BPFC` would be virtual functions are implemented: they call their truly virtual counterparts in `BPFCImpl`. The result is a functor class (`BPFC`) that's small and monomorphic, yet has access to a large amount of state and acts polymorphically.

I'm glossing over a fair number of details here, because the basic technique I've sketched is well known in C++ circles. *Effective C++* treats it in Item 34. In *Design Patterns* by Gamma *et al*. [6], this is called the "Bridge Pattern." Sutter calls this the "Pimpl Idiom" in his *Exceptional C++* [8].

From an STL point of view, the primary thing to keep in mind is that functor classes using this technique must support copying in a reasonable fashion. If you were the author of `BPFC` above, you'd have to make sure that its copy constructor did something reasonable about the `BPFCImpl` object it points to. Perhaps the simplest reasonable thing would be to reference count it, using something like Boost's `shared_ptr`, which you can read about in Item 50.

In fact, for purposes of this Item, the *only* thing you'd have to worry about would be the behavior of `BPFC`'s copy constructor, because function objects are always copied—passed by value, remember?—when passed to or returned from functions in the STL. That means two things. Make them small, and make them monomorphic.

Item 39: Make predicates pure functions

I hate to do this to you, but we have to start with a short vocabulary lesson.

- A *predicate* is a function that returns `bool` (or something that can be implicitly converted to `bool`). Predicates are widely used in the STL. The comparison functions for the standard associative containers are predicates, and predicate functions are commonly passed as parameters to algorithms like `find_if` and the various sorting algorithms. (For an overview of the sorting algorithms, turn to Item 31.)
- A *pure function* is a function whose return value depends only on its parameters. If `f` is a pure function and `x` and `y` are objects, the return value of `f(x, y)` can change only if the value of `x` or `y` changes.

  In C++, all data consulted by pure functions are either passed in as parameters or are constant for the life of the function. (Naturally, such constant data should be declared `const`.) If a pure function consulted data that might change between calls, invoking the function at different times with the same parameters might yield different results, and that would be contrary to the definition of a pure function.

That should be enough to make it clear what it means to make predicates pure functions. All I have to do now is convince you that the advice is well founded. To help me do that, I hope you'll forgive me for burdening you with one more term.

- A *predicate class* is a functor class whose `operator()` function is a predicate, i.e., its `operator()` returns `true` or `false` (or something that can be implicitly converted to `true` or `false`). As you might expect, any place the STL expects a predicate, it will accept either a real predicate or an object of a predicate class.

That's it, I promise! Now we're ready to study why this Item offers guidance worth following.

Item 38 explains that function objects are passed by value, so you should design function objects to be copied. For function objects that are predicates, there is another reason to design them to behave well when they are copied. Algorithms may make copies of functors and hold on to them a while before using them, and some algorithm implementations take advantage of this freedom. A critical repercussion of this observation is that *predicate functions must be pure functions*.

To appreciate why this is the case, let's suppose you were to violate this constraint. Consider the following (badly designed) predicate class. Regardless of the arguments that are passed, it returns `true` exactly once: the third time it is called. The rest of the time it returns `false`.

Click here to view code image

```
class BadPredicate:                              // see Item
40 for info
  public unary_function<Widget, bool> {          // on the
base class
public:
  BadPredicate(): timesCalled(0) {}              // init
timesCalled to 0

  bool operator()(const Widget&)
  {
    return ++timesCalled == 3;
  }

private:
  size_t timesCalled;
};
```

Suppose we use this class to eliminate the third `Widget` from a `vector<Widget>`:

Click here to view code image

```
vector<Widget> vw;                       // create vector and
put some
...                                      // Widgets into it

vw.erase(remove_if(vw.begin(),           // eliminate the
third Widget;
                   vw.end(),             // see Item 32 for
info on how
                   BadPredicate()),      // erase and
remove_if relate
```

```
          vw.end());
```

This code looks quite reasonable, but with many STL implementations, it will eliminate not just the third element from `vw`, it will also eliminate the sixth!

To understand how this can happen, it's helpful to see how `remove_if` is often implemented. Bear in mind that `remove_if` does not *have* to be implemented this way.

Click here to view code image

```
template <typename FwdIterator, typename Predicate>
FwdIterator remove_if(FwdIterator begin, FwdIterator end,
Predicate p)
{

  begin = find_if(begin, end, p);

  if (begin == end) return begin;
  else {
    FwdIterator next = begin;
    return remove_copy_if(++next, end, begin, p);
  }
}
```

The details of this code are not important, but note that the predicate `p` is passed first to `find_if`, then later to `remove_copy_if`. In both cases, of course, `p` is passed by value—is *copied*—into those algorithms. (Technically, this need not be true, but practically, it *is* true. For details, see Item 38.)

The initial call to `remove_if` (the one in the client code that wants to eliminate the third element from `vw`) creates an anonymous `BadPredicate` object, one with its internal `timesCalled` member set to zero. This object (known as `p` inside `remove_if`) is then copied into `find_if`, so `find_if` also receives a `BadPredicate` object with a `timesCalled` value of 0. `find_if` "calls" that object until it returns `true`, so it calls it three times. `find_if` then returns control to `remove_if`. `remove_if` continues to execute and ultimately calls `remove_copy_if`, passing as a predicate another copy of `p`. But `p`'s `timesCalled` member is still 0! `find_if` never called `p`, it called only a *copy* of `p`. As a result, the third time `remove_copy_if` calls its predicate, it, too, will return `true`. And that's why `remove_if` will ultimately remove two Widgets from `vw` instead of just one.

The easiest way to keep yourself from tumbling into this linguistic crevasse is to declare your `operator()` functions `const` in predicate classes. If you do that, your compilers won’t let you change any class data members:

Click here to view code image

```
class BadPredicate:
  public unary_function<Widget, bool> {
public:

  bool operator()(const Widget&) const
  {
    return ++timesCalled == 3;           // error! can't
change local data
  }                                      // in a const member
function

  ...
};
```

Because this is such a straightforward way of preventing the problem we just examined, I very nearly entitled this Item “Make `operator()` `const` in predicate classes.” But that doesn’t go far enough. Even `const` member functions may access `mutable` data members, non-`const` local `static` objects, non-`const` class `static` objects, non-`const` objects at namespace scope, and non-`const` global objects. A well-designed predicate class ensures that its `operator()` functions are independent of those kinds of objects, too. Declaring `operator()` `const` in predicate classes is *necessary* for correct behavior, but it’s not *sufficient*. A well-behaved `operator()` is certainly `const`, but it’s more than that. It’s also a pure function.

Earlier in this Item, I remarked that any place the STL expects a predicate function, it will accept either a real function or an object of a predicate class. That’s true in both directions. Any place the STL will accept an object of a predicate class, a predicate function (possibly modified by `ptr_fun`—see Item 41) is equally welcome. We now understand that `operator()` functions in predicate classes should be pure functions, so this restriction extends to predicate functions, too. This function is as bad a predicate as the objects generated from the `BadPredicate` class:

Click here to view code image

```
bool anotherBadPredicate(const Widget&, const Widget&)
{
  static int timesCalled = 0;    // No! No! No! No! No! No!
No!
```

```
  return ++timesCalled == 3;     // Predicates should be pure
functions,
}                                // and pure functions have
no state
```

Regardless of how you write your predicates, they should always be pure functions.

Item 40: Make functor classes adaptable

Suppose I have a list of `Widget*` pointers and a function to determine whether such a pointer identifies a `Widget` that is interesting:

```
list<Widget*> widgetPtrs;

bool isInteresting(const Widget *pw);
```

If I'd like to find the first pointer to an interesting `Widget` in the list, it'd be easy:

Click here to view code image

```
list<Widget*>::iterator i = find_if(widgetPtrs.begin(),
widgetPtrs.end(),
                                    isInteresting);
if (i != widgetPtrs.end()) {
    ...                                                  //
process the first
}                                                        // ptr
to an interesting
                                                         //
Widget
```

If I'd like to find the first pointer to a `Widget` that is *not* interesting, however, the obvious approach fails to compile:

Click here to view code image

```
list<Widget*>::iterator i =
   find_if(widgetPtrs.begin(), widgetPtrs.end(),
           not1(isInteresting));                         //
error! won't compile
```

Instead, I must apply `ptr_fun` to `isInteresting` before applying `not1`:

Click here to view code image

```
list<Widget*>::iterator i =
   find_if(widgetPtrs.begin(), widgetPtrs.end(),
            not1(ptr_fun(isInteresting)));              // fine

if (i != widgetPtrs.end()) {
    ...                                                 //
process the first ptr
}                                                       // to a
dull Widget
```

That leads to some questions. *Why* do I have to apply `ptr_fun` to `isInteresting` before applying `not1`? What does `ptr_fun` do for me, and how does it make the above work?

The answer is somewhat surprising. The only thing `ptr_fun` does is make some typedefs available. That's it. These typedefs are required by `not1`, and that's why applying `not1` to `ptr_fun` works, but applying `not1` to `isInteresting` directly doesn't work. Being a lowly function pointer, `isInteresting` lacks the typedefs that `not1` demands.

`not1` isn't the only component in the STL making such demands. Each of the four standard function adapters (`not1`, `not2`, `bind1st`, and `bind2nd`) requires the existence of certain typedefs, as do any nonstandard STL-compatible adapters written by others (e.g., those available from SGI and Boost—see Item 50). Function objects that provide the necessary typedefs are said to be *adaptable*, while function objects lacking these typedefs are not adaptable. Adaptable function objects can be used in more contexts than can function objects that are not adaptable, so you should make your function objects adaptable whenever you can. It costs you nothing, and it may buy clients of your functor classes a world of convenience.

I know, I know, I'm being coy, constantly referring to "certain typedefs" without telling you what they are. The typedefs in question are `argument_type`, `first_argument_type`, `second_argument_type`, and `result_type`, but it's not quite that straightforward, because different kinds of functor classes are expected to provide different subsets of these names. In all honesty, unless you're writing your own adapters (a topic not covered in this book), you don't need to know anything about these typedefs. That's because the conventional way to provide them is to inherit them from a base class, or, more precisely, a base struct. For functor classes whose `operator()` takes one argument, the struct to inherit from is `std::unary_function`. For functor classes whose `operator()` takes two arguments, the struct to inherit from is `std::binary_function`.

Well, sort of. `unary_function` and `binary_function` are templates, so you can't inherit from them directly. Instead, you must inherit from structs they generate, and that requires that you specify some type arguments. For `unary_function`, you must specify the type of parameter taken by your functor class's `operator()`, as well as its return type. For `binary_function`, you specify three types: the types of your `operator()`'s first and second parameters, and your `operator()`'s return type.

Here are a couple of examples:

Click here to view code image

```
template<typename T>
class MeetsThreshold: public std::unary_function<Widget,
bool> {
private:
  const T threshold;

public:
  MeetsThreshold(const T& threshold);

  bool operator()(const Widget&) const;

  ...
};

struct WidgetNameCompare:
  public std::binary_function<Widget, Widget, bool> {

  bool operator()(const Widget& lhs, const Widget& rhs)
const;
};
```

In both cases, notice how the types passed to `unary_function` or `binary_function` are the same as the types taken and returned by the functor class's `operator()`, though it is a bit of an oddity that `operator()`'s return type is passed as the last argument to `unary_function` or `binary_function`.

You may have noticed that `MeetsThreshold` is a class, while `WidgetNameCompare` is a struct. `MeetsThreshold` has internal state (its `threshold` data member), and a class is the logical way to encapsulate such information. `WidgetNameCompare` has no state, hence no need to make anything private. Authors of functor classes where everything is public often declare structs instead of classes, probably for no other reason than to avoid typing "`public`" in front of the base class and the `operator()` function. Whether to declare such functors as classes or structs is purely a matter of personal style. If you’re still refining your personal style and would like to emulate the pros, note that stateless functor classes within the STL itself (e.g., `less<T>`, `plus<T>`, etc.) are generally written as structs.

Look again at `WidgetNameCompare`:

Click here to view code image

```
struct WidgetNameCompare:
  public std::binary_function<Widget, Widget, bool> {

  bool operator()(const Widget& lhs, const Widget& rhs)
const;
};
```

Even though `operator()`’s arguments are of type `const Widget&`, the type passed to `binary_function` is `Widget`. In general, non-pointer types passed to `unary_function` or `binary_function` have `const`s and references stripped off. (Don’t ask why. The reasons are neither terribly good nor terribly interesting. If you’re dying to know anyway, write some programs where you don’t strip them off, then dissect the resulting compiler diagnostics. If, having done that, you’re *still* interested in the matter, visit boost.org (see Item 50) and check out their work on call traits and function object adapters.)

The rules change when `operator()` takes pointer parameters. Here’s a struct analogous to `WidgetNameCompare`, but this one works with `Widget*` pointers:

Click here to view code image

```
struct PtrWidgetNameCompare:
  public std::binary_function<const Widget*, const Widget*,
bool> {

  bool operator()(const Widget* lhs, const Widget* rhs)
const;
};
```

Here, the types passed to `binary_function` are the *same* as the types taken by `operator()`. The general rule for functor classes taking or returning pointers is to pass to `unary_function` or `binary_function` whatever types `operator()` takes or returns.

Let’s not forget the fundamental reason for all this `unary_function` and `binary_function` base class gobbledegook. These classes supply typedefs that are required by function object adapters, so inheritance from those classes yields adaptable function objects. That lets us do things like this:

[Click here to view code image]

```
list<Widget> widgets;

...

list<Widget>::reverse_iterator i1 =            // find the
last widget
   find_if(widgets.rbegin(), widgets.rend(),   // that fails
to meet the
           not1(MeetsThreshold<int>(10)));     // threshold
of 10
                                               // (whatever
that means)

Widget w( constructor arguments );

list<Widget>::iterator i2 =                    // find the
first widget
   find_if(widgets.begin(), widgets.end(),     // that
precedes w in the
           bind2nd(WidgetNameCompare(), w));   // sort order
defined by
                                               //
WidgetNameCompare
```

Had we failed to have our functor classes inherit from `unary_function` or `binary_function`, neither of these examples would compile, because `not1` and `bind2nd` both work only with adaptable function objects.

STL function objects are modeled on C++ functions, and a C++ function has only one set of parameter types and one return type. As a result, the STL implicitly assumes that each functor class has only one `operator()` function, and it's the parameter and return types for this function that should be passed to `unary_function` or `binary_function` (in accord with the rules for reference and pointer types we just discussed). This means that, tempting though it might be, you shouldn't try to combine the functionality of `WidgetNameCompare` and `PtrWidgetNameCompare` by creating a single struct with two `operator()` functions. If you did, the functor would be adaptable with respect to at most one of its calling forms (whichever one you used when passing parameters to `binary_function`), and a functor that's adaptable only half the time might just as well not be adaptable at all.

Sometimes it makes sense to give a functor class multiple invocation forms (thereby abandoning adaptability), and Items 7, 20, 23, and 25 give examples of situations where that is the case. Such functor classes are the exception, however, not the rule. Adaptability is important, and you should strive to facilitate it each time you write a functor class.

Item 41: Understand the reasons for `ptr_fun`, `mem_fun`, and `mem_fun_ref`

What *is* it with this `ptr_fun`/`mem_fun`/`mem_fun_ref` stuff? Sometimes you have to use these functions, sometimes you don't, and what do they do, anyway? They just seem to sit there, pointlessly hanging around function names like ill-fitting garments. They're unpleasant to type, annoying to read, and resistant to comprehension. Are these things additional examples of STL artifacts (such as the ones described in Items 10 and 18), or just some syntactic joke foisted on us by members of a Standardization Committee with too much free time and a twisted sense of humor?

Calm yourself. The names are less than inspired, but `ptr_fun`, `mem_fun`, and `mem_fun_ref` do important jobs, and as far as syntactic jokes go, one of the primary tasks of these functions is to paper over one of C++'s inherent syntactic inconsistencies.

If I have a function `f` and an object `x`, I wish to invoke `f` on `x`, and I'm outside `x`'s member functions, C++ gives me three different syntaxes for making the call:

Click here to view code image

```
f(x);                                   // Syntax #1: When f
is a
                                        // non-member
function

x.f();                                  // Syntax #2: When f
is a member
                                        // function and x is
an object or
                                        // a reference to an
object

p->f();                                 // Syntax #3: When f
is a member
```

```
                                        // function and p is
a pointer to x
```

Now, suppose I have a function that can test `Widget`s,

[Click here to view code image]

```
void test(Widget& w);                   // test w and mark it
as "failed" if
                                        // it doesn't pass
the test
```

and I have a container of `Widget`s:

```
vector<Widget> vw;                      // vw holds widgets
```

To test every `Widget` in `vw`, I can use `for_each` in the obvious manner:

```
for_each(vw.begin(), vw.end(), test);   // Call #1 (compiles)
```

But imagine that `test` is a member function of `Widget` instead of a non-member function, i.e., that `Widget` supports self-testing:

[Click here to view code image]

```
class Widget {
public:
  ...
  void test();                          // perform a self-
test; mark *this
  ...                                   // as "failed" if it
doesn't pass
};
```

In a perfect world, I'd also be able to use `for_each` to invoke `Widget::test` on each object in `vw`:

[Click here to view code image]

```
for_each(vw.begin(), vw.end(),
         &Widget::test);                // Call #2 (won't
compile)
```

In fact, if the world were really perfect, I'd be able to use `for_each` to invoke `Widget::test` on a container of `Widget*` pointers, too:

[Click here to view code image]

```
list<Widget*> lpw;                      // lpw holds pointers
to widgets

for_each(lpw.begin(), lpw.end(),
         &Widget::test);                 // Call #3 (also
won't compile)
```

But think of what would have to happen in this perfect world. Inside the `for_each` function in Call #1, we'd be calling a non-member function with an object, so we'd have to use Syntax #1. Inside the `for_each` function in Call #2, we'd have to use Syntax #2, because we'd have an object and a member function. And inside the `for_each` function in Call #3, we'd need to use Syntax #3, because we'd be dealing with a member function and a pointer to an object. We'd therefore need three different versions of `for_each`, and how perfect would that world be?

In the world we do have, we possess only one version of `for_each`. It's not hard to envision an implementation:

Click here to view code image

```
template<typename InputIterator, typename Function>
Function for_each(InputIterator begin, InputIterator end,
Function f)
{
  while (begin != end) f(*begin++);
}
```

Here I've highlighted the fact that `for_each` uses Syntax #1 when making the call. This is a universal convention in the STL: functions and function objects are always invoked using the syntactic form for non-member functions. This explains why Call #1 compiles while Calls #2 and #3 don't. It's because STL algorithms (including `for_each`) hardwire in Syntax #1, and only Call #1 is compatible with that syntax.

Perhaps it's now clear why `mem_fun` and `mem_fun_ref` exist. They arrange for member functions (which must ordinarily be called using Syntax #2 or #3) to be called using Syntax #1.

The way `mem_fun` and `mem_fun_ref` do this is simple, though it's a little clearer if you take a look at a declaration for one of these functions. They're really function templates, and several variants of the `mem_fun` and `mem_fun_ref` templates exist, corresponding to different numbers of parameters and the `const`ness (or lack thereof) of the member functions they adapt. Seeing one declaration is enough to understand how things are put together:

Click here to view code image

```
template<typename R, typename C>     // declaration for
```

```
mem_fun for
mem_fun_t<R,C>                       // non-const member
functions
mem_fun(R (C::*pmf)());              // taking no parameters.
C is the
                                     // class, R is the
return type of the
                                     // pointed-to member
function
```

`mem_fun` takes a pointer to a member function, `pmf`, and returns an object of type `mem_fun_t`. This is a functor class that holds the member function pointer and offers an `operator()` that invokes the pointed-to member function on the object passed to `operator()`. For example, in this code,

Click here to view code image

```
list<Widget*> lpw;                   // same as above

...

for_each(lpw.begin(), lpw.end(),
         mem_fun(&Widget::test));    // this will now compile
```

`for_each` receives an object of type `mem_fun_t` holding a pointer to `Widget::test`. For each `Widget*` pointer in `lpw`, `for_each` “calls” the `mem_fun_t` object using Syntax #1, and that object immediately invokes `Widget::test` on the `Widget*` pointer using Syntax #3.

Overall, `mem_fun` adapts Syntax #3, which is what `Widget::test` requires when used with a `Widget*` pointer, to Syntax #1, which is what `for_each` uses. It’s thus no wonder that classes like `mem_fun_t` are known as *function object adapters*. It should not surprise you to learn that, completely analogously with the above, the `mem_fun_ref` functions adapt Syntax #2 to Syntax #1 and generate adapter objects of type `mem_fun_ref_t`.

The objects produced by `mem_fun` and `mem_fun_ref` do more than allow STL components to assume that all functions are called using a single syntax. They also provide important typedefs, just like the objects produced by `ptr_fun`. The story behind these typedefs is told in Item 40, so I won’t repeat it here. However, this puts us in a position to understand why this call compiles,

```
for_each(vw.begin(), vw.end(), test);              // as
above, Call #1;
```

```
                                                     // this
compiles
```

while these do not:

Click here to view code image

```
for_each(vw.begin(), vw.end(), &Widget::test);      // as
above, Call #2;
                                                     // doesn't
compile

for_each(lpw.begin(), lpw.end(), &Widget::test);    // as
above, Call #3;
                                                     // doesn't
compile
```

The first call (Call #1) passes a real function, so there's no need to adapt its calling syntax for use by `for_each`; the algorithm will inherently call it using the proper syntax. Furthermore, `for_each` makes no use of the typedefs that `ptr_fun` adds, so it's not necessary to use `ptr_fun` when passing `test` to `for_each`. On the other hand, adding the typedefs can't hurt anything, so this will do the same thing as the call above:

Click here to view code image

```
for_each(vw.begin(), vw.end(), ptr_fun(test));      //
compiles and behaves
                                                     // like
Call #1 above
```

If you get confused about when to use `ptr_fun` and when not to, consider using it every time you pass a function to an STL component. The STL won't care, and there is no runtime penalty. About the worst that can be said is that some people reading your code might raise an eyebrow when they see an unnecessary use of `ptr_fun`. How much that bothers you depends, I suppose, on your sensitivity to raised eyebrows.

An alternative strategy with respect to `ptr_fun` is to use it only when you're forced to. If you omit it when the typedefs are necessary, your compilers will balk at your code. Then you'll have to go back and add it.

The situation with `mem_fun` and `mem_fun_ref` is fundamentally different. You must employ them whenever you pass a member function to an STL component, because, in addition to adding typedefs (which may or may not be necessary), they adapt the calling syntaxes from the ones normally used with member functions to the one used everywhere in the STL. If you don't use them when passing member function pointers, your code will never compile.

Which leaves just the names of the member function adapters, and here, finally, we have a genuine historical STL artifact. When the need for these kinds of adapters first became apparent, the people working on the STL focused on containers of pointers. (Given the drawbacks of such containers described in Items 7, 20, and 33, this might seem surprising, but remember that containers of pointers support polymorphism, while containers of objects do not.) They needed an adapter for member functions, so they chose `mem_fun`. Only later did they realize that they needed a different adapter for containers of objects, so they hacked up the name `mem_fun_ref` for that. No, it's not very elegant, but these things happen. Tell me *you've* never given any of your components a name that you later realized was, er, difficult to generalize... .

Item 42: Make sure `less<T>` means `operator<`

As all `Widget`-savvy people are aware, `Widget`s have both a weight and a maximum speed:

Click here to view code image

```
class Widget {
public:
  ...

  size_t weight() const;
  size_t maxSpeed() const;

  ...
};
```

Also well known is that the natural way to sort `Widget`s is by weight. `operator<` for `Widget`s reflects this:

Click here to view code image

```
bool operator<(const Widget& lhs, const Widget& rhs)
{
  return lhs.weight() < rhs.weight();
}
```

But suppose we'd like to create a `multiset<Widget>` where the `Widget`s are sorted by maximum speed. We know that the default comparison function for `multiset<Widget>` is `less<Widget>`, and we know that `less<Widget>`, by default, does its work by calling `operator<` for `Widget`s. That being the case, it seems clear that one way to get a `multiset<Widget>` sorted by maximum speed is to sever the tie between `less<Widget>` and `operator<` by specializing `less<Widget>` to look only at a `Widget`’s maximum speed:

Click here to view code image

```
template<>                                    // This is a
specialization
struct std::less<Widget>:                     // of std::less
for Widget;
  public                                      // it's also a
very bad idea
  std::binary_function<Widget,
                       Widget,                // See Item 40
for info
                       bool> {                // on this base
class

  bool operator()(const Widget& lhs, const Widget& rhs) const
  {
    return lhs.maxSpeed() < rhs.maxSpeed();
  }
};
```

This both looks ill-advised and is ill-advised, but it may not be ill-advised for the reason you think. Does it surprise you that it compiles at all? Many programmers point out that the above isn’t just a specialization of a template, it’s a specialization of a template in the `std` namespace. “Isn’t `std` supposed to be sacred, reserved for library implementers and beyond the reach of mere programmers?” they ask. “Shouldn’t compilers reject this attempt to tamper with the workings of the C++ immortals?” they wonder.

As a general rule, trying to modify components in `std` is indeed forbidden (and doing so typically transports one to the realm of undefined behavior), but under some circumstances, tinkering is allowed. Specifically, programmers are allowed to specialize templates in `std` for user-defined types. Almost always, there are alternatives that are superior to specializing `std` templates, but on rare occasions, it’s a reasonable thing to do. For instance, authors of smart pointer classes often want their classes to act like built-in pointers for sorting purposes, so it’s not uncommon to see specializations of `std::less` for smart pointer types. The following, for example, is part of the Boost library’s `shared_ptr`, a smart pointer you can read about in Items 7 and 50:

Click here to view code image

```
namespace std {

  template<typename T>                       // this is a
spec. of std::less
  struct less< boost::shared_ptr<T> >:       // for
boost::shared_ptr<T>
    public                                   // (boost is a
namespace)
    binary_function<boost::shared_ptr<T>,
                    boost::shared_ptr<T>,    // this is the
customary
                    bool> {                  // base class
(see Item 40)

    bool operator()(const boost::shared_ptr<T>& a,
                    const boost::shared_ptr<T>& b) const
    {
      return less<T*>()(a.get(),b.get());    //
shared_ptr::get returns
    }                                        // the built-in
pointer that's
                                             // in the
shared_ptr object
  };
}
```

This isn’t unreasonable, and it certainly serves up no surprises, because this specialization of `less` merely ensures that sorting smart pointers behaves the same as sorting their built-in brethren. Alas, our tentative specialization of `less` for `Widget` *does* serve up surprises.

C++ programmers can be forgiven certain assumptions. They assume that copy constructors copy, for example. (As Item 8 attests, failure to adhere to this convention can lead to astonishing behavior.) They assume that taking the address of an object yields a pointer to that object. (Turn to Item 18 to read about what can happen when this isn’t true.) They assume that adapters like `bind1st` and `not2` may be applied to function objects. (Item 40 explains how things break when this isn’t the case.) They assume that `operator+` adds (except for `string`s, but there’s a long history of using “+” to mean string concatenation), that `operator-` subtracts, that `operator==` compares. And they assume that using `less` is tantamount to using `operator<`.

`operator<` is more than just the default way to implement `less`, it's *what programmers expect* `less` *to do*. Having `less` do something other than call `operator<` is a gratuitous violation of programmers' expectations. It runs contrary to what has been called "the principle of least astonishment." It's callous. It's mean. It's bad. You shouldn't do it.

Especially when there's no reason to. There's not a place in the STL using `less` where you can't specify a different comparison type instead. Returning to our original example of a `multiset<Widget>` ordered by maximum speed, all we need to do to get what we want is create a functor class called almost anything *except* `less` that performs the comparison we're interested in. Why, here's one now:

Click here to view code image

```
struct MaxSpeedCompare:
  public binary_function<Widget, Widget, bool> {

  bool operator()(const Widget& lhs, const Widget& rhs) const
  {
    return lhs.maxSpeed() < rhs.maxSpeed();
  }
};
```

To create our `multiset`, we use `MaxSpeedCompare` as the comparison type, thus avoiding use of the default comparison type (which is, of course, `less<Widget>`):

```
multiset<Widget, MaxSpeedCompare> widgets;
```

This code says exactly what it means. It creates a `multiset` of `Widget`s sorted as defined by the functor class `MaxSpeedCompare`.

Contrast that with this:

```
multiset<Widget> widgets;
```

This says that `widgets` is a `multiset` of `Widget`s sorted in the default manner. Technically, that means it uses `less<Widget>`, but virtually everybody is going to assume that really means it's sorted by `operator<`.

Don't mislead all those programmers by playing games with the definition of `less`. If you use `less` (explicitly or implicitly), make sure it means `operator<`. If you want to sort objects using some other criterion, create a special functor class that’s *not* called `less`. It’s really as simple as that.

Chapter 7. Programming with the STL

It's traditional to summarize the STL as consisting of containers, iterators, algorithms, and function objects, but *programming* with the STL is much more than that. Programming with the STL is knowing when to use loops, when to use algorithms, and when to use container member functions. It's knowing when `equal_range` is a better way to search than `lower_bound`, knowing when `lower_bound` is preferable to `find`, and knowing when `find` beats `equal_range`. It's knowing how to improve algorithm performance by substituting functors for functions that do the same thing. It's knowing how to avoid unportable or incomprehensible code. It's even knowing how to read compiler error messages that run to thousands of characters. And it's knowing about Internet resources for STL documentation, STL extensions, even complete STL implementations.

Yes, programming with the STL involves knowing many things. This chapter gives you much of the knowledge you need.

Item 43: Prefer algorithm calls to hand-written loops

Every algorithm takes at least one pair of iterators that specify a range of objects over which to do something. `min_element` finds the smallest value in the range, for example, while `accumulate` summarizes some information about the range as a whole (see Item 37) and `partition` separates all the elements of a range into those that do and do not satisfy some criterion (see Item 31). For algorithms to do their work, they must examine every object in the range(s) they are passed, and they do this in the way you'd expect: they loop from the beginning of the range(s) to the end. Some algorithms, such as `find` and `find_if`, may return before they complete the traversal, but even these algorithms internally contain a loop. After all, even `find` and `find_if` must look at every element of a range before they can conclude that what they are looking for is *not* present.

Internally, then, algorithms are loops. Furthermore, the breadth of STL algorithms means that many tasks you might naturally code as loops could also be written using algorithms. For example, if you have a `Widget` class that supports redrawing,

```
class Widget {
public:
  ...
  void redraw() const;
  ...
};
```

and you’d like to redraw all the `Widget`s in a `list`, you could do it with a loop, like this,

Click here to view code image

```
list<Widget> lw;

...

for (list<Widget>::iterator i = lw.begin(); i != lw.end();
++i) {
  i->redraw();
}
```

but you could also do it with the `for_each` algorithm:

Click here to view code image

```
for_each(lw.begin(), lw.end(),                    // see Item 41
for info
         mem_fun_ref(&Widget::redraw));           // on
mem_fun_ref
```

For many C++ programmers, writing the loop is more natural than calling the algorithm, and reading the loop is more comfortable than making sense of `mem_fun_ref` and the taking of `Widget::redraw`’s address. Yet this Item argues that the algorithm call is preferable. In fact, this Item argues that calling an algorithm is usually preferable to *any* hand-written loop. Why?

There are three reasons:

- **Efficiency:** Algorithms are often more efficient than the loops programmers produce.
- **Correctness:** Writing loops is more subject to errors than is calling algorithms.
- **Maintainability:** Algorithm calls often yield code that is clearer and more straightforward than the corresponding explicit loops.

The remainder of this Item lays out the case for algorithms.

From an efficiency perspective, algorithms can beat explicit loops in three ways, two major, one minor. The minor way involves the elimination of redundant computations. Look again at the loop we just saw:

Click here to view code image

```
for (list<Widget>::iterator i = lw.begin(); i != lw.end();
++i) {
  i->redraw();
}
```

I've highlighted the loop termination test to emphasize that each time around the loop, `i` will be checked against `lw.end()`. That means that each time around the loop, the function `list::end` will be invoked. But we don't need to call `end` more than once, because we're not modifying the list. A single call to `end` would suffice, and, if we look again at the algorithm invocation, we'll see that that's exactly how many times `end` is evaluated:

```
for_each(lw.begin(), lw.end(),                  // this call
evaluates
         mem_fun_ref(&Widget::redraw));         // lw.end()
exactly once
```

To be fair, STL implementers understand that `begin` and `end` (and similar functions, such as `size`) are used frequently, so they're likely to design them for maximal efficiency. They'll almost certainly `inline` them and strive to code them so that most compilers will be able to avoid repeated computations by hoisting their results out of loops like the one above. Experience shows that implementers don't always succeed, however, and when they don't, the avoidance of repeated computations is enough to give the algorithm a performance edge over the hand-written loop.

But that's the minor efficiency argument. The first major argument is that library implementers can take advantage of their knowledge of container implementations to optimize traversals in a way that no library user ever could. For example, the objects in a `deque` are typically stored (internally) in one or more fixed-size arrays. Pointer-based traversals of these arrays are faster than iterator-based traversals, but only library implementers can use pointer-based traversals, because only they know the size of the internal arrays and how to move from one array to the next. Some STLs contain algorithm implementations that take their `deque`'s internal data structures into account, and such implementations have been known to clock in at more than 20% faster than the "normal" implementations of the algorithms.

The point is not that STL implementations are optimized for `deque`s (or any other specific container type), but that implementers know more about their implementations than you do, and they can take advantage of this knowledge in algorithm implementations. If you shun algorithm calls in favor of your own loops, you forgo the opportunity to benefit from any implementation-specific optimizations they may have provided.

The second major efficiency argument is that all but the most trivial STL algorithms use computer science algorithms that are more sophisticated—sometimes *much* more sophisticated—than anything the average C++ programmer will be able to come up with. It’s next to impossible to beat `sort` or its kin (see Item 31); the search algorithms for sorted ranges (see Items 34 and 45) are equally good; and even such mundane tasks as eliminating some objects from contiguous-memory containers are more efficiently accomplished using the `erase-remove` idiom than the loops most programmers come up with (see Item 9).

If the efficiency argument for algorithms doesn’t persuade you, perhaps you’re more amenable to a plea based on correctness. One of the trickier things about writing your own loops is making sure you use only iterators that (a) are valid and (b) point where you want them to. For example, suppose you have an array (presumably due to a legacy C API—see Item 16), and you’d like to take each array element, add 41 to it, then insert it into the front of a deque. Writing your own loop, you might come up with this (which is a variant on an example from Item 16):

Click here to view code image

```
// C API: this function takes a pointer to an array of at
most arraySize
// doubles and writes data to it. It returns the number of
doubles written.
size_t fillArray(double *pArray, size_t arraySize);

double data[maxNumDoubles];                            // create
local array of
                                                       // max
possible size

deque<double> d;                                       // create
deque, put
...                                                    // data
```

```
into it

size_t numDoubles =
   fillArray(data, maxNumDoubles);                  // get
array data from API

for (size_t i = 0; i < numDoubles; ++i) {          // for
each i in data,
   d.insert(d.begin(), data[i] + 41);               // insert
data[i]+41 at the
}                                                  // front
of d; this code
                                                   // has a
bug!
```

This works, as long as you’re happy with a result where the newly inserted elements are in the reverse order of the corresponding elements in `data`. Because each insertion location is `d.begin()`, the last element inserted will go at the front of the deque!

If that’s not what you wanted (and admit it, it’s not), you might think to fix it like this:

Click here to view code image

```
deque<double>::iterator insertLocation = d.begin();   //
remember d's
                                                      //
begin iterator

for (size_t i = 0; i < numDoubles; ++i) {       // insert
data[i]+41
   d.insert(insertLocation++, data[i] + 41);     // at
insertLocation, then
}                                               // increment
                                                //
insertLocation; this
                                                // code is
also buggy!
```

This looks like a double win, because it not only increments the iterator specifying the insertion position, it also eliminates the need to call `begin` each time around the loop; that eliminates the minor efficiency hit we discussed earlier. Alas, this approach runs into a different problem: it yields undefined results. Each time `deque::insert` is called, it invalidates all iterators into the `deque`, and that includes `insertLocation`. After the first call to `insert`, `insertLocation` is invalidated, and subsequent loop iterations are allowed to head straight to looneyland.

Once you puzzle this out (possibly with the aid of STLport's debug mode, which is described in Item 50), you might come up with the following:

Click here to view code image

```
deque<double>::iterator insertLocation =
  d.begin();                                        // as
before

for (size_t i = 0; i < numDoubles; ++i) {           // update
insertLocation
  insertLocation =                                  // each
time insert is
    d.insert(insertLocation, data[i] + 41);         // called
to keep the
  ++insertLocation;                                 // iterator
valid, then
}                                                   //
increment it
```

This code finally does what you want, but think about how much work it took to get here! Compare that to the following call to `transform`:

Click here to view code image

```
transform(data, data + numDoubles,                   // copy
all elements
          inserter(d, d.begin()),                    // from
data to the front
          bind2nd(plus<double>(), 41));              // of d,
adding 41 to each
```

The "`bind2nd(plus<double>(), 41)`" might take you a couple of minutes to get right (especially if you don't use STL's binders very often), but the only iterator-related worries you have are specifying the beginning and end of the source range (which was never a problem) and being sure to use `inserter` as the beginning of the destination range (see Item 30). In practice, figuring out the correct initial iterators for source and destination ranges is usually easy, or at least a lot easier than making sure the body of a loop doesn't inadvertently invalidate an iterator you need to keep using.

This example is representative of a broad class of loops that are difficult to write correctly, because you have to be on constant alert for iterators that are incorrectly manipulated or are invalidated before you're done using them. To see a different example of how inadvertent iterator invalidation can lead to trouble, turn to Item 9, which describes the subtleties involved in writing loops that call `erase`.

Given that using invalidated iterators leads to undefined behavior, and given that undefined behavior has a nasty habit of failing to show itself during development and testing, why run the risk if you don't have to? Turn the iterators over to the algorithms, and let *them* worry about the vagaries of iterator manipulation.

I've explained why algorithms can be more efficient than hand-written loops, and I've described why such loops must navigate a thicket of iterator-related difficulties that algorithms avoid. With luck, you are now an algorithm believer. Yet luck is fickle, and I'd prefer a more secure conviction before I rest my case. Let us therefore move on to the issue of code clarity. In the long run, the best software is the clearest software, the software that is easiest to understand, the software that can most readily be enhanced, maintained, and molded to fit new circumstances. The familiarity of loops notwithstanding, algorithms have an advantage in this long-term competition.

The key to their edge is the power of a known vocabulary. There are 70 algorithm names in the STL—a total of over 100 different function templates, once overloading is taken into account. Each of those algorithms carries out some well-defined task, and *it is reasonable to expect professional C++ programmers to know (or be able to look up) what each does*. Thus, when a programmer sees a `transform` call, that programmer recognizes that some function is being applied to every object in a range, and the results of those calls are being written somewhere. When the programmer sees a call to `replace_if`, he or she knows that all the objects in a range that satisfy some predicate are being modified. When the programmer comes across an invocation of `partition`, she or he understands that the objects in a range are being moved around so that all the objects satisfying a predicate are grouped together (see Item 31). The names of STL algorithms convey a lot of semantic information, and that makes them clearer than any random loop can hope to be.

When you see a `for`, `while`, or `do`, all you know is that some kind of loop is coming up. To acquire even the faintest idea of what that loop does, you have to examine it. Not so with algorithms. Once you see a call to an algorithm, the name alone sketches the outline of what it does. To understand exactly what will happen, of course, you must inspect the arguments being passed to the algorithm, but that's often less work than trying to divine the intent of a general looping construct.

Simply put, algorithm names suggest what they do. "`for`," "`while`," and "`do`" don't. In fact, this is true of any component of the standard C or C++ library. Without doubt, you could write your own implementations of `strlen`, `memset`, or `bsearch`, if you wanted to, but you don't. Why not? Because (1) somebody has already written them, so there's no point in your doing it again; (2) the names are standard, so everybody knows what they do; and (3) you suspect that your library implementer knows some efficiency tricks you don't know, and you're unwilling to give up the possible optimizations a skilled library implementer might provide. Just as you don't write your own versions of `strlen` *et al.*, it makes no sense to write loops that duplicate functionality already present in STL algorithms.

I wish that were the end of the story, because I think it's a strong finish. Alas, this is a tale that refuses to go gentle into that good night.

Algorithm names are more meaningful than bare loops, it's true, but specifying what to do during an iteration can be clearer using a loop than using an algorithm. For example, suppose you'd like to identify the first element in a vector whose value is greater than some `x` and less than some `y`. Here's how you could do it using a loop:

Click here to view code image

```
vector<int> v;
int x, y;
...

vector<int>::iterator i = v.begin();       // iterate from
v.begin() until an
for( ; i != v.end(); ++i) {                // appropriate
value is found or
  if (*i > x && *i < y) break;             // v.end() is
reached
}
...                                        // i now points to
the value or is
                                           // the same as
```

```
v.end()
```

It is possible to pass this same logic to `find_if`, but it requires that you use a nonstandard function object adapter like SGI's `compose2` (see Item 50):

Click here to view code image

```
vector<int>::iterator i =
  find_if(v.begin(), v.end(),                          // find the
first value val
          compose2(logical_and<bool>(),                // where
the "and" of
                   bind2nd(greater<int>(), x),         // val> x
and
                   bind2nd(less<int>(), y)));          // val < y
                                                       // is true
```

Even if this didn't use nonstandard components, many programmers would object that it's nowhere near as clear as the loop, and I have to admit to being sympathetic to that view (see Item 47).

The `find_if` call can be made less imposing by moving the test logic into a separate functor class,

Click here to view code image

```
template<typename T>
class BetweenValues:
  public unary_function<T, bool> {                  // see Item 40
public:
  BetweenValues(const T& lowValue,
                const T& highValue)                 // have the
ctor save the
  : lowVal(lowValue), highVal(highValue)            // values to
be between
  {}

  bool operator()(const T& val) const               // return
whether
  {                                                 // val is
between the
    return val > lowVal && val < highVal;           // saved
values
  }

private:
  T lowVal;
  T highVal;
};

...

vector<int>::iterator i = find_if(v.begin(), v.end(),
                                  BetweenValues<int>(x, y));
```

but this has its own drawbacks. First, creating the `BetweenValues` template is a lot more work than writing the loop body. Just count the lines. Loop body: one; `BetweenValues` template: fourteen. Not a very good ratio. Second, the details of what `find_if` is looking for are now physically separate from the call. To really understand the call to `find_if`, one must look up the definition of `BetweenValues`, but `BetweenValues` must be defined outside the function containing the call to `find_if`. If you try to declare `BetweenValues` *inside* the function containing the call to `find_if`, like this,

[Click here to view code image](#)

```
{                                                  //
beginning of function
    ...

    template <typename T>
    class BetweenValues: public unary_function<T, bool> { ...
};

    vector<int>::iterator i = find_if(v.begin(), v.end(),
                                      BetweenValues<int>(x,
y));
    ...
}                                                  // end of
function
```

you'll discover that it won't compile, because templates can't be declared inside functions. If you try to avoid that restriction by making `BetweenValues` a class instead of a template,

[Click here to view code image](#)

```
{                                                  //
beginning of function
    ...

    class BetweenValues: public unary_function<int, bool> {
... };
```

```
    vector<int>::iterator i = find_if(v.begin(), v.end(),
                                   BetweenValues(x, y));
    ...
}                                                  // end of
function
```

you'll find that you're still out of luck, because classes defined inside functions are known as *local classes*, and local class types can't be bound to template type arguments (such as the type of the functor taken by `find_if`). Sad as it may seem, functor classes and functor class templates are not allowed to be defined inside functions, no matter how convenient it would be to be able to do it.

In the ongoing tussle between algorithm calls and hand-written loops, the bottom line on code clarity is that it all depends on what you need to do inside the loop. If you need to do something an algorithm already does, or if you need to do something very similar to what an algorithm does, the algorithm call is clearer. If you need a loop that does something fairly simple, but would require a confusing tangle of binders and adapters or would require a separate functor class if you were to use an algorithm, you're probably better off just writing the loop. Finally, if you need to do something fairly long and complex inside the loop, the scales tilt back toward algorithms, because long, complex computations should generally be moved into separate functions, anyway. Once you've moved the loop body into a separate function, you can almost certainly find a way to pass that function to an algorithm (often `for_each`) such that the resulting code is direct and straightforward.

If you agree with this Item that algorithm calls are generally preferable to hand-written loops, and if you also agree with Item 5 that range member functions are preferable to loops that iteratively invoke single-element member functions, an interesting conclusion emerges: well-crafted C++ programs using the STL contain far fewer loops than equivalent programs not using the STL. This is a good thing. Any time we can replace low-level words like `for`, `while`, and `do` with higher-level terms like `insert`, `find`, and `for_each`, we raise the level of abstraction in our software and thereby make it easier to write, document, enhance, and maintain.

Item 44: Prefer member functions to algorithms with

the same names

Some containers have member functions with the same names as STL algorithms. The associative containers offer `count`, `find`, `lower_bound`, `upper_bound`, and `equal_range`, while `list` offers `remove`, `remove_if`, `unique`, `sort`, `merge`, and `reverse`. Most of the time, you’ll want to use the member functions instead of the algorithms. There are two reasons for this. First, the member functions are faster. Second, they integrate better with the containers (especially the associative containers) than do the algorithms. That’s because algorithms and member functions that share the same name typically do *not* do exactly the same thing.

We’ll begin with an examination of the associative containers. Suppose you have a `set<int>` holding a million values and you’d like to find the first occurrence of the value 727, if there is one. Here are the two most obvious ways to perform the search:

Click here to view code image

```
set<int> s;                                                  //
create set, put
...                                                          //
1,000,000 values
                                                             //
into it

set<int>::iterator i = s.find(727);                          //
use find member
if (i != s.end()) ...                                        //
function

set<int>::iterator i = find(s.begin(), s.end(), 727);        //
use find algorithm
if (i != s.end()) ...
```

The `find` member function runs in logarithmic time, so, regardless of whether 727 is in the set, `set::find` will perform no more than about 40 comparisons looking for it, and usually it will require only about 20. In contrast, the `find` algorithm runs in linear time, so it will have to perform 1,000,000 comparisons if 727 isn’t in the set. Even if 727 is in the set, the `find` algorithm will perform, on average, 500,000 comparisons to locate it. The efficiency score is thus

Member `find`: About 40 (worst case) to about 20 (average case)

Algorithm `find`: 1,000,000 (worst case) to 500,000 (average case)

As in golf, the low score wins, and as you can see, this matchup is not much of a contest.

I have to be a little cagey about the number of comparisons required by member `find`, because it's partially dependent on the implementation used by the associative containers. Most implementations use red-black trees, a form of balanced tree that may be out of balance by up to a factor of two. In such implementations, the maximum number of comparisons needed to search a set of a million values is 38, but for the vast majority of searches, no more than 22 comparisons are required. An implementation based on perfectly balanced trees would never require more than 21 comparisons, but in practice, the overall performance of such perfectly balanced trees is inferior to that of red-black trees. That's why most STL implementations use red-black trees.

Efficiency isn't the only difference between member and algorithm `find`. As Item 19 explains, STL algorithms determine whether two objects have the same value by checking for equality, while associative containers use equivalence as their "sameness" test. Hence, the `find` algorithm searches for 727 using equality, while the `find` member function searches using equivalence. The difference between equality and equivalence can be the difference between a successful search and an unsuccessful search. For example, Item 19 shows how using the `find` algorithm to look for something in an associative container could fail even when the corresponding search using the `find` member function would succeed! You should therefore prefer the member form of `find`, `count`, `lower_bound`, etc., over their algorithm eponyms when you work with associative containers, because they offer behavior that is consistent with the other member functions of those containers. Due to the difference between equality and equivalence, algorithms don't offer such consistent behavior.

This difference is especially pronounced when working with `map`s and `multimap`s, because these containers hold `pair` objects, yet their member functions look only at the key part of each pair. Hence, the `count` member function counts only pairs with matching keys (a "match," naturally, is determined by testing for equivalence); the value part of each pair is ignored.

The member functions `find`, `lower_bound`, `upper_bound`, and `equal_range` behave similarly. If you use the `count` algorithm, however, it will look for matches based on (a) equality and (b) both components of the pair; `find`, `lower_bound`, etc., do the same thing. To get the algorithms to look at only the key part of a pair, you have to jump through the hoops described in Item 23 (which would also allow you to replace equality testing with equivalence testing).

On the other hand, if you are really concerned with efficiency, you may decide that Item 23's gymnastics, in conjunction with the logarithmic-time lookup algorithms of Item 34, are a small price to pay for an increase in performance. Then again, if you're *really* concerned with efficiency, you'll want to consider the non-standard hashed containers described in Item 25, though there you'll again confront the difference between equality and equivalence.

For the standard associative containers, then, choosing member functions over algorithms with the same names offers several benefits. First, you get logarithmic-time instead of linear-time performance. Second, you determine whether two values are "the same" using equivalence, which is the natural definition for associative containers. Third, when working with `map`s and `multimap`s, you automatically deal only with key values instead of with (key, value) pairs. This triumvirate makes the case for preferring member functions pretty iron-clad.

Let us therefore move on to `list` member functions that have the same names as STL algorithms. Here the story is almost completely about efficiency. Each of the algorithms that `list` specializes (`remove`, `remove_if`, `unique`, `sort`, `merge`, and `reverse`) copies objects, but `list`-specific versions copy nothing; they simply manipulate the pointers connecting list nodes. The algorithmic complexity of the algorithms and the member functions is the same, but, under the assumption that manipulating pointers is less expensive than copying objects, `list`'s versions of these functions should offer better performance.

It's important to bear in mind that the `list` member functions often behave differently from their algorithm counterparts. As Item 32 explains, calls to the algorithms `remove`, `remove_if`, and `unique` must be followed by calls to `erase` if you really want to eliminate objects from a container, but `list`'s `remove`, `remove_if`, and `unique` member functions honestly get rid of elements; no subsequent call to `erase` is necessary.

A significant difference between the `sort` algorithm and `list`'s `sort` function is that the former can't be applied to `list`s. Being only bidirectional iterators, `list`'s iterators can't be passed to `sort`. A gulf also exists between the behavior of the `merge` algorithm and `list`'s `merge`. The algorithm isn't permitted to modify its source ranges, but `list::merge` *always* modifies the lists it works on.

So there you have it. When faced with a choice between an STL algorithm or a container member function with the same name, you should prefer the member function. It's almost certain to be more efficient, and it's likely to be better integrated with the container's usual behavior, too.

Item 45: Distinguish among `count`, `find`, `binary_search`, `lower_bound`, `upper_bound`, and `equal_range`

So you want to look for something, and you have a container or you have iterators demarcating a range where you think it's located. How do you conduct the search? Your quiver is fairly bursting with arrows: `count`, `count_if`, `find`, `find_if`, `binary_search`, `lower_bound`, `upper_bound`, and `equal_range`. Decisions, decisions! How do you choose?

Easy. You reach for something that's fast and simple. The faster and simpler, the better.

For the time being, we'll assume that you have a pair of iterators specifying a range to be searched. Later, we'll consider the case where you have a container instead of a range.

In selecting a search strategy, much depends on whether your iterators define a sorted range. If they do, you can get speedy (usually logarithmic-time—see Item 34) lookups via `binary_search`, `lower_bound`, `upper_bound`, and `equal_range`. If the iterators don't demarcate a sorted range, you're limited to the linear-time algorithms `count`, `count_if`, `find`, and `find_if`. In what follows, I'll ignore the `_if` variants of `count` and `find`, just as I'll ignore the variants of `binary_search`, `lower_` and `upper_bound`, and `equal_range` taking a predicate. Whether you rely on the default search predicate or you specify your own, the considerations for choosing a search algorithm are the same.

If you have an unsorted range, your choices are `count` or `find`. They answer slightly different questions, so it's worth taking a closer look at them. `count` answers the question, “Is the value there, and if so, how many copies are there?” while `find` answers the question, “Is it there, and if so, where is it?”

Suppose all you want to know is whether some special `Widget` value `w` is in a `list`. Using `count`, the code looks like this:

[Click here to view code image]

```
list<Widget> lw;                          // list of Widgets
Widget w;                                 // special Widget
value

...

if (count(lw.begin(), lw.end(), w)) {
   ...                                    // w is in lw
} else {
   ...                                    // it's not
}
```

This demonstrates a common idiom: using `count` as an existence test. `count` returns either zero or a positive number, so we rely on the conversion of nonzero values to `true` and zero to `false`. It would arguably be clearer to be more explicit about what we are doing,

```
if (count(lw.begin(), lw.end(), w) != 0) ...
```

and some programmers do write it that way, but it’s quite common to rely on the implicit conversion, as in the original example.

Compared to that original code, using `find` is slightly more complicated, because you have to test `find`’s return value against the list’s end iterator:

[Click here to view code image]

```
if (find(lw.begin(), lw.end(), w) != lw.end()) {
   ...
} else {
   ...
}
```

For existence testing, the idiomatic use of `count` is slightly simpler to code. At the same time, it’s also less efficient when the search is successful, because `find` stops once it’s found a match, while `count` must continue to the end of the range looking for additional matches. For most programmers, `find`’s edge in efficiency is enough to justify the slight increase in usage complexity.

Often, knowing whether a value is in a range isn't enough. Instead, you'll want to know the first object in the range with the value. For example, you might want to print the object, you might want to insert something in front of it, or you might want to erase it (but see Item 9 for guidance on erasing while iterating). When you need to know not just whether a value exists but also which object (or objects) has that value, you need `find`:

Click here to view code image

```
list<Widget>::iterator i = find(lw.begin(), lw.end(), w);

if (i != lw.end()) {

   ...                                        // found it, i
points to the first one

} else {
   ...                                        // didn't find it
}
```

For sorted ranges, you have other choices, and you'll definitely want to use them. `count` and `find` run in linear time, but the search algorithms for sorted ranges (`binary_search`, `lower_bound`, `upper_bound`, and `equal_range`) run in logarithmic time.

The shift from unsorted ranges to sorted ranges leads to another shift: from using equality to determine whether two values are the same to using equivalence. Item 19 comprises a discourse on equality versus equivalence, so I won't repeat it here. Instead, I'll simply note that the `count` and `find` algorithms both search using equality, while `binary_search`, `lower_bound`, `upper_bound`, and `equal_range` employ equivalence.

To test for the existence of a value in a sorted range, use `binary_search`. Unlike `bsearch` in the standard C library (and hence also in the standard C++ library), `binary_search` returns only a `bool`: whether the value was found. `binary_search` answers the question, "Is it there?," and its answer is either yes or no. If you need more information than that, you need a different algorithm.

Here's an example of `binary_search` applied to a sorted vector. (You can read about the virtues of sorted vectors in Item 23.)

Click here to view code image

```
vector<Widget> vw;                                       // create
vector, put
```

```
...                                                     // data
into it, sort the
sort(vw.begin(), vw.end());                             // data

Widget w;                                               // value
to search for
...

if (binary_search(vw.begin(), vw.end(), w)) {
   ...                                                  // w is in
vw
} else {
   ...                                                  // it's
not
}
```

If you have a sorted range and your question is, “Is it there, and if so, where is it?” you want `equal_range`, but you may think you want `lower_bound`. We’ll discuss `equal_range` shortly, but first, let’s examine `lower_bound` as a way of locating values in a range.

When you ask `lower_bound` to look for a value, it returns an iterator pointing to either the first copy of that value (if it’s found) or to the proper insertion location for that value (if it’s not). `lower_bound` thus answers the question, “Is it there? If so, where is the first copy, and if it’s not, where would it go?” As with `find`, the result of `lower_bound` must be tested to see if it’s pointing to the value you’re looking for. Unlike `find`, you can’t just test `lower_bound`’s return value against the end iterator. Instead, you must test the object `lower_bound` identifies to see if that’s the value you want.

Many programers use `lower_bound` like this:

[Click here to view code image]

```
vector<Widget>::iterator i = lower_bound(vw.begin(),
vw.end(), w);

if (i != vw.end() && *i == w) {         // make sure i points
to an object;
                                        // make sure the object
has the
                                        // correct value; this
has a bug!

   ...                                  // found the value, i
points to the
                                      // first object with
that value
} else {
   ...                                // not found
}
```

This works most of the time, but it’s not really correct. Look again at the test to determine whether the desired value was found:

```
if (i != vw.end() && *i == w  ) ...
```

This is an *equality* test, but `lower_bound` searched using *equivalence*. Most of the time, tests for equivalence and equality yield the same results, but Item 19 demonstrates that it’s not that hard to come up with situations where equality and equivalence are different. In such situations the code above is wrong.

To do things properly, you must check to see if the iterator returned from `lower_bound` points to an object with a value that is equivalent to the one you searched for. You could do that manually (Item 19 shows you how, and Item 24 provides an example of when it can be worthwhile), but it can get tricky, because you have to be sure to use the same comparison function that `lower_bound` used. In general, that could be an arbitrary function (or function object). If you passed a comparison function to `lower_bound`, you’d have to be sure to use the same comparison function in your hand-coded equivalence test. *That* would mean that if you changed the comparison function you passed to `lower_bound`, you’d have to make the corresponding change in your check for equivalence. Keeping the comparison functions in sync isn’t rocket science, but it is another thing to remember, and I suspect you already have plenty you’re expected to keep in mind.

There is an easier way: use `equal_range`. `equal_range` returns a *pair* of iterators, the first equal to the iterator `lower_bound` would return, the second equal to the one `upper_bound` would return (i.e., the one-past-the-end iterator for the range of values equivalent to the one searched for). `equal_range`, then, returns a pair of iterators that demarcate the range of values equivalent to the one you searched for. A well-named algorithm, no? (`equivalent_range` would be better, of course, but `equal_range` is still pretty good.)

There are two important observations about `equal_range`’s return value. First, if the two iterators are the same, that means the range of objects is empty; the value wasn’t found. That observation is the key to using `equal_range` to answer the question, “Is it there?” You use it like this:

Click here to view code image

```
vector<Widget> vw;
...
sort(vw.begin(), vw.end());

typedef vector<Widget>::iterator VWIter;  // convenience
typedefs
typedef pair<VWIter, VWIter> VWIterPair;

VWIterPair p = equal_range(vw.begin(), vw.end(), w);

if (p.first != p.second) {                // if equal_range
didn't return
                                          // an empty
range...

   ...                                    // found it,
p.first points to the
                                          // first one and
p.second
                                          // points to one
past the last
} else {


   ...                                    // not found, both
p.first and
                                          // p.second point
to the
}                                         // insertion
location for
                                          // the value
searched for
```

This code uses only equivalence, so it is always correct.

The second thing to note about `equal_range`’s return value is that the distance between its iterators is the number of objects in the range, i.e., the objects with a value equivalent to the target of the search. For types where equality and equivalence yield the same results, then, `equal_range` does the job of both `find` and `count` for sorted ranges. To locate the `Widget`s in `vw` with a value equivalent to `w` and then print out how many such `Widget`s exist, for example, you could do this:

Click here to view code image

```
VWIterPair p = equal_range(vw.begin(), vw.end(), w);

cout << "There are " << distance(p.first, p.second)
     << " elements in vw equivalent to w.";
```

So far, our discussion has assumed we want to search for a value in a range, but sometimes we're more interested in finding a *location* in a range. For example, suppose we have a `Timestamp` class and a `vector` of `Timestamp`s that's sorted so that older timestamps come first:

Click here to view code image

```
class Timestamp { ... };

bool operator<(const Timestamp& lhs,         // returns
whether lhs
               const Timestamp& rhs);        // precedes rhs
in time

vector<Timestamp> vt;                        // create
vector, fill it with
...                                          // data, sort it
so that older
sort(vt.begin(), vt.end());                  // times precede
newer ones
```

Now suppose we have a special timestamp, `ageLimit`, and we want to remove from `vt` all the timestamps that are older than `ageLimit`. In this case, we don't want to search `vt` for a `Timestamp` equivalent to `ageLimit`, because there might not be any elements with that exact value. Instead, we need to find a *location* in `vt`: the first element that is no older than `ageLimit`. This is as easy as easy can be, because `lower_bound` will give us precisely what we need:

Click here to view code image

```
Timestamp ageLimit;
...

vt.erase(vt.begin(), lower_bound(vt.begin(),      //
eliminate from vt all
                                 vt.end(),        // objects
that precede
                                 ageLimit));      //
```

```
ageLimit's value
```

If our requirements change slightly so that we want to eliminate all the timestamps that are at least as old as `ageLimit`, we need to find the location of the first timestamp that is *younger* than `ageLimit`. That's a job tailor-made for `upper_bound`:

[Click here to view code image](#)

```
vt.erase(vt.begin(), upper_bound(vt.begin(),        //
eliminate from vt all
                                 vt.end(),          // objects
that precede
                                 ageLimit));        // or are
equivalent
                                                    // to
ageLimit's value
```

`upper_bound` is also useful if you want to insert things into a sorted range so that objects with equivalent values are stored in the order in which they were inserted. For example, we might have a sorted `list` of `Person` objects, where the objects are sorted by name:

[Click here to view code image](#)

```
class Person {
public:
  ...
  const string& name() const;
  ...
};

struct PersonNameLess:
  public binary_function<Person, Person, bool> {       // see
Item 40

  bool operator()(const Person& lhs, const Person& rhs) const
  {
    return lhs.name() < rhs.name();
  }
};

list<Person> lp;
...

lp.sort(PersonNameLess());                             //
sort lp using
```

```
                                                    //
PersonNameLess
```

To keep the list sorted the way we desire (by name, with equivalent names stored in the order in which they are inserted), we can use `upper_bound` to specify the insertion location:

Click here to view code image

```
Person newPerson;

...

lp.insert( upper_bound(lp.begin(),            // insert
newPerson after
                       lp.end(),              // the last
object in lp
                       newPerson,             // that
precedes or is
                       PersonNameLess()),     // equivalent
to
           newPerson);                        // newPerson
```

This works fine and is quite convenient, but it’s important not to be misled by this use of `upper_bound` into thinking that we’re magically looking up an insertion location in a `list` in logarithmic time. We’re not. Item 34 explains that because we’re working with a `list`, the lookup takes linear time, but it performs only a logarithmic number of comparisons.

Up to this point, we have considered only the case where you have a pair of iterators defining a range to be searched. Often you have a container, not a range. In that case, you must distinguish between the sequence and associative containers. For the standard sequence containers (`vector`, `string`, `deque`, and `list`), you follow the advice we’ve outlined in this Item, using the containers’ begin and end iterators to demarcate the range.

The situation is different for the standard associative containers (`set`, `multiset`, `map`, and `multimap`), because they offer member functions for searching that are generally better choices than the STL algorithms. Item 44 goes into the details of why they are better choices, but briefly, it’s because they’re faster and they behave more naturally. Fortunately, the member functions usually have the same names as the corresponding algorithms, so where the foregoing discussion recommends you choose algorithms named `count`, `find`, `equal_range`, `lower_bound`, or `upper_bound`, you simply select the same-named member functions when searching associative containers. `binary_search` calls for a different strategy, because there is no member function analogue to this algorithm. To test for the existence of a value in a `set` or `map`, use `count` in its idiomatic role as a test for membership:

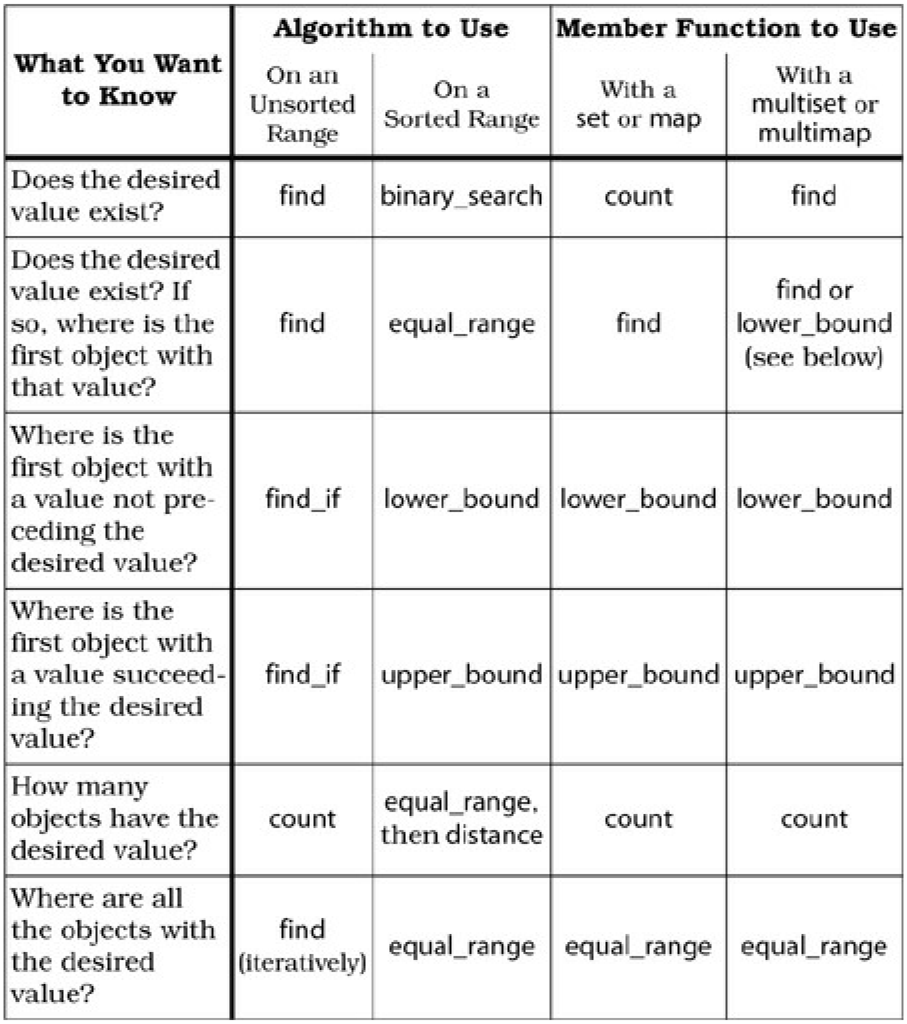
Click here to view code image

```
set<Widget> s;                      // create set, put data
into it
...

Widget w;                           // w still holds the
value to search for
...

if (s.count(w)) {

    ...                             // a value equivalent to
w exists
} else {
    ...                             // no such value exists
}
```

To test for the existence of a value in a `multiset` or `multimap`, `find` is generally superior to `count`, because `find` can stop once it's found a single object with the desired value, while `count`, in the worst case, must examine every object in the container. (This isn't a problem when working with `set`s and `map`s, because `set`s don't allow duplicate values and `map`s don't allow duplicate keys.)

However, `count`'s role for counting things in associative containers is secure. In particular, it's a better choice than calling `equal_range` and applying `distance` to the resulting iterators. For one thing, it's clearer: `count` means "count." For another, it's easier; there's no need to create a pair and pass its components to `distance`. For a third, it's probably a little faster.

Given everything we've considered in this Item, where do we stand? The following table says it all.



What You Want to Know	Algorithm to Use		Member Function to Use	
	On an Unsorted Range	On a Sorted Range	With a set or map	With a multiset or multimap
Does the desired value exist?	find	binary_search	count	find
Does the desired value exist? If so, where is the first object with that value?	find	equal_range	find	find or lower_bound (see below)
Where is the first object with a value not preceding the desired value?	find_if	lower_bound	lower_bound	lower_bound
Where is the first object with a value succeeding the desired value?	find_if	upper_bound	upper_bound	upper_bound
How many objects have the desired value?	count	equal_range, then distance	count	count
Where are all the objects with the desired value?	find (iteratively)	equal_range	equal_range	equal_range

In the column summarizing how to work with sorted ranges, the frequency with which `equal_range` occurs may be surprising. That frequency arises from the importance of testing for equivalence when searching. With `lower_bound` and `upper_bound`, it's too easy to fall back on equality tests, but with `equal_range`, testing only for equivalence is the natural thing to do. In the second row for sorted ranges, `equal_range` beats out `find` for an additional reason: `equal_range` runs in logarithmic time, while `find` takes linear time.

For `multiset`s and `multimap`s, the table lists both `find` and `lower_bound` as candidates when you're looking for the first object with a particular value. `find` is a common choice for this job, and you may have noticed that it's the one listed in the table for `set`s and `map`s. For the `multi` containers, however, `find` is not guaranteed to identify the *first* element in the container with a given value if more than one is present; its charter is only to identify *one* of those elements. If you really need to find the *first* object with a given value, you'll want to employ `lower_bound`, and you'll have to manually perform the second half of the equivalence test described in Item 19 to confirm that you've found the value you were looking for. (You could avoid the manual equivalence test by using `equal_range`, but calling `equal_range` is more expensive than calling `lower_bound`.)

Selecting among `count`, `find`, `binary_search`, `lower_bound`, `upper_bound`, and `equal_range` is easy. Choose the algorithm or member function that offers you the behavior and performance you need and that requires the least amount of work when you call it. Follow that advice (or consult the table), and you should never get confused.

Item 46: Consider function objects instead of functions as algorithm parameters

One of the complaints about programming in high-level languages is that as the level of abstraction gets higher, the efficiency of the generated code gets lower. In fact, Alexander Stepanov (the inventor of the STL) once produced a small benchmark suite that tried to measure the "abstraction penalty" of C++ vis-à-vis C. Among other things, the results of that benchmark revealed that it's nearly universal for the code generated for manipulating a class containing a `double` to be less efficient than the corresponding code for manipulating a `double` directly. It may thus come as a surprise to learn that passing STL function objects—objects masquerading as functions—to algorithms typically yields code that is *more* efficient than passing real functions.

For example, suppose you need to sort a `vector` of `double`s in descending order. The straightforward STL way to do it is via the `sort` algorithm and a function object of type `greater<double>`:

Click here to view code image

```
vector<double> v;

...

sort(v.begin(), v.end(), greater<double>());
```

If you're wary of the abstraction penalty, you might decide to eschew the function object in favor of a real function, a function that's not only real, it's `inline`:

Click here to view code image

```
inline
bool doubleGreater(double d1, double d2)
{
  return d1 > d2;
}

...

sort(v.begin(), v.end(), doubleGreater);
```

Interestingly, if you were to compare the performance of the two calls to `sort` (one using `greater<double>`, one using `doubleGreater`), you'd almost certainly find that the one using `greater<double>` was faster. For instance, I timed the two calls to `sort` on a `vector` of a million `double`s using four different STL platforms, each set to optimize for speed, and the version using `greater<double>` was faster every time. At worst, it was 50% faster, at best it was 160% faster. So much for the abstraction penalty.

The explanation for this behavior is simple: inlining. If a function object's `operator()` function has been declared inline (either explicitly via `inline` or implicitly by defining it in its class definition), the body of that function is available to compilers, and most compilers will happily inline that function during template instantiation of the called algorithm. In the example above, `greater<double>::operator()` is an inline function, so compilers inline-expand it during instantiation of `sort`. As a result, `sort` contains zero function calls, and compilers are able to perform optimizations on this call-free code that are otherwise not usually attempted. (For a discussion of the interaction between inlining and compiler optimization, see Item 33 of *Effective C++* and chapters 8–10 of Bulka and Mayhew's *Efficient C++* [10].)

The situation is different for the call to `sort` using `doubleGreater`. To see how it's different, we must recall that there's no such thing as passing a function as a parameter to another function. When we try to pass a function as a parameter, compilers silently convert the function into a *pointer* to that function, and it's the pointer we actually pass. Hence, the call

```
sort(v.begin(), v.end(), doubleGreater);
```

doesn't pass `doubleGreater` to `sort`, it passes a pointer to `doubleGreater`. When the `sort` template is instantiated, this is the declaration for the function that is generated:

Click here to view code image

```
void sort(vector<double>::iterator first,      // beginning
of range
          vector<double>::iterator last,       // end of
range
          bool (*comp)(double, double));       // comparison
function
```

Because `comp` is a pointer to a function, each time it's used inside `sort`, compilers make an indirect function call—a call through a pointer. Most compilers won't try to inline calls to functions that are invoked through function pointers, even if, as in this example, such functions have been declared `inline` and the optimization appears to be straightforward. Why not? Probably because compiler vendors have never felt that it was worthwhile to implement the optimization. You have to have a little sympathy for compiler vendors. They have lots of demands on their time, and they can't do everything. Not that this should stop you from asking them for it.

The fact that function pointer parameters inhibit inlining explains an observation that long-time C programmers often find hard to believe: C++'s `sort` virtually always embarrasses C's `qsort` when it comes to speed. Sure, C++ has function and class templates to instantiate and funny-looking `operator()` functions to invoke while C makes a simple function call, but all that C++ "overhead" is absorbed during compilation. At runtime, `sort` makes inline calls to its comparison function (assuming the comparison function has been declared `inline` and its body is available during compilation) while `qsort` calls its comparison function through a pointer. The end result is that `sort` runs faster. In my tests on a `vector` of a million doubles, it ran up to 670% faster, but don't take my word for it, try it yourself. It's easy to verify that when comparing function objects and real functions as algorithm parameters, there's an abstraction *bonus*.

There's another reason to prefer function objects to functions as algorithm parameters, and it has nothing to do with efficiency. It has to do with getting your programs to compile. For whatever reason, it's not uncommon for STL platforms to reject perfectly valid code, either through shortcomings in the compiler or the library or both. For example, one widely used STL platform rejects the following (valid) code to print to `cout` the length of each string in a set:

Click here to view code image

```
set<string> s;

...

transform(s.begin(), s.end(),
          ostream_iterator<string::size_type>(cout, "\n"),
          mem_fun_ref(&string::size));
```

The cause of the problem is that this particular STL platform has a bug in its handling of `const` member functions (such as `string::size`). A workaround is to use a function object instead:

Click here to view code image

```
struct StringSize:
  public unary_function<string, string::size_type> {      //
see Item 40

  string::size_type operator()(const string& s) const
  {
    return s.size();
  }
};

transform(s.begin(), s.end(),
          ostream_iterator<string::size_type>(cout, "\n"),
          StringSize());
```

There are other workarounds for this problem, but this one does more than just compile on every STL platform I know. It also facilitates inlining the call to `string::size`, something that would almost certainly not take place in the code above where `mem_fun_ref(&string::size)` is passed to `transform`. In other words, creation of the functor class `StringSize` does more than sidestep compiler conformance problems, it's also likely to lead to an increase in performance.

Another reason to prefer function objects to functions is that they can help you avoid subtle language pitfalls. Occasionally, source code that looks reasonable is rejected by compilers for legitimate, but obscure, reasons. There are situations, for example, when the name of an instantiation of a function template is not equivalent to the name of a function. Here's one such situation:

[Click here to view code image]

```
template<typename FPType>                                    //
return the average
FPType average(FPType val1, FPType val2)                     //
of 2 floating point
{                                                            //
numbers
  return (val1 + val2) / 2;
}

template<typename InputIter1,
         typename InputIter2>
void writeAverages(InputIter1 begin1,                        //
write the pairwise
                   InputIter1 end1,                          //
averages of 2
                   InputIter2 begin2,                        //
sequences to a
                   ostream& s)                               //
stream
{
  transform(
    begin1, end1, begin2,
    ostream_iterator<typename
iterator_traits<InputIter1>::value_type>(s, "\n"),


    average<typename
iterator_traits<InputIter1>::value_type>  // error?
  );
}
```

Many compilers accept this code, but the C++ Standard appears to forbid it.[†] The reasoning is that there could, in theory, be another function template named `average` that takes a single type parameter. If there were, the expression `average<typename iterator_traits<InputIter1>:: value_type>` would be ambiguous, because it would not be clear which template to instantiate. In this particular example, no ambiguity is present, but some compilers reject the code anyway, and they are allowed to do that. No matter. The solution to the problem is to fall back on a function object:

[†] In October 2003, the Standardization Committee clarified that this code should be valid. This decision doesn't officially take effect until the next

version of the C++ standard is adopted (anticipated to be 2009), but compiler vendors are likely to implement the revised rules sooner.

Click here to view code image

```
template<typename FPType>
struct Average:
  public binary_function<FPType, FPType, FPType> {       //
see Item 40

  FPType operator()(FPType val1, FPType val2) const
  {
    return average(val1, val2);
  }
};

template<typename InputIter1, typename InputIter2>
void writeAverages(InputIter1 begin1, InputIter1 end1,
                   InputIter2 begin2, ostream& s)
{
  transform(
    begin1, end1, begin2,
    ostream_iterator<typename
iterator_traits<InputIter1>::value_type>(s, "\n"),
    Average<typename iterator_traits<InputIter1>::value_type>
()
  );
}
```

Every compiler should accept this revised code. Furthermore, calls to `Average::operator()` inside `transform` are eligible for inlining, something that would not be true for an instantiation of `average` above, because `average` is a template for functions, not function *objects*.

Function objects as parameters to algorithms thus offer more than greater efficiency. They're also more robust when it comes to getting your code to compile. Real functions are useful, of course, but when it comes to effective STL programing, function objects are frequently more useful.

Item 47: Avoid producing write-only code

Suppose you have a `vector<int>`, and you'd like to get rid of all the elements in the vector whose value is less than `x` and that occur after the last value at least as big as `y`. Does the following instantly spring to mind?

Click here to view code image

```
vector<int> v;
int x, y;

...

v.erase(
  remove_if(find_if(v.rbegin(), v.rend(),
                    bind2nd(greater_equal<int>(), y)).base(),
            v.end(),
            bind2nd(less<int>(), x)),
  v.end());
```

One statement, and the job is done. Clear and straightforward. No problem. Right?

Well, let's step back for a moment. Does this strike you as reasonable, maintainable code? "No!" shriek most C++ programmers, fear and loathing in their voices. "Yes!" squeal a few, delight evident in theirs. And therein lies the problem. One programmer's vision of expressive purity is another programmer's demonic missive from Hell.

As I see it, there are two causes for concern in the code above. First, it's a rat's nest of function calls. To see what I mean, here is the same statement, but with all the function names replaced by f*n*, each *n* corresponding to one of the functions:

```
v.f1(f2(f3(v.f4(), v.f5(), f6(f7(), y)).f8(), v.f9(),
f6(f10(), x)), v.f9());
```

This looks unnaturally complicated, because I've removed the indentation present in the original example, but I think it's safe to say that any statement involving twelve function calls to ten different functions would be considered excessive by most C++ software developers. Programmers weaned on functional languages such as Scheme might feel differently, however, and my experience has been that the majority of programmers who view the original code without raising an eyebrow have a strong functional programming background. Most C++ programmers lack this background, so unless your colleagues are versed in the ways of deeply nested function calls, code like the `erase` call above is almost sure to confound the next person who is forced to make sense of what you have written.

The second drawback of the code is the significant STL background needed to understand it. It uses the less common `_if` forms of `find` and `remove`, it uses reverse iterators (see Item 26), it converts `reverse_iterator`s to `iterator`s (see Item 28), it uses `bind2nd`, it creates anonymous function objects, and it employs the `erase-remove` idiom (see Item 32). Experienced STL programmers can swallow that combination without difficulty, but far more C++ developers will have their eyes glaze over before they've taken so much as a bite. If your colleagues are well-steeped in the ways of the STL, using `erase`, `remove_if`, `find_if`, `base`, and `bind2nd` in a single statement may be fine, but if you want your code to be comprehensible by C++ programmers with a more mainstream background, I encourage you to break it down into more easily digestible chunks.

Here's one way you could do it. (The comments aren't just for this book. I'd put them in the code, too.)

Click here to view code image

```
typedef vector<int>::iterator VecIntIter;

// Initialize rangeBegin to point to the element following
the last
// occurrence of a value greater than or equal to y. If there
is no such
// value, initialize rangeBegin to v.begin(). If the last
occurrence of the
// value is the last element in v, initialize rangeBegin to
v.end().
VecIntIter rangeBegin = find_if(v.rbegin(), v.rend(),
                                bind2nd(greater_equal<int>(),
y)).base();

// from rangeBegin to v.end(), erase everything with a value
less than x
v.erase(remove_if(rangeBegin, v.end(), bind2nd(less<int>(),
x)), v.end());
```

This is still likely to confuse some people, because it relies on an understanding of the `erase-remove` idiom, but between the comments in the code and a good STL reference (e.g., Josuttis' *The C++ Standard Library* [3] or *SGI's STL web site* [21]), every C++ programmer should be able to figure out what's going on without too much difficulty.

When transforming the code, it's important to note that I didn't abandon algorithms and try to write my own loops. Item 43 explains why that's generally an inferior option, and its arguments apply here. When writing source code, the goal is to come up with code that is meaningful to both compilers and humans and that offers acceptable performance. Algorithms are almost always the best way to achieve that goal. However, Item 43 also explains how the increased use of algorithms naturally leads to an increased tendency to nest function calls and to throw in binders and other functor adapters. Look again at the problem specification that opened this Item:

> Suppose you have a `vector<int>`, and you'd like to get rid of all the elements in the vector whose value is less than `x` and that occur after the last value at least as big as `y`.

The outline of a solution *does* spring to mind:

- Finding the last occurrence of a value in a vector calls for some application of `find` or `find_if` with reverse iterators.
- Getting rid of elements calls for either `erase` or the `erase-remove` idiom.

Put those two ideas together, and you get this pseudocode, where "something" indicates a placeholder for code that hasn't yet been fleshed out:

Click here to view code image

```
v.erase(remove_if(find_if(v.rbegin(), v.rend(),
something).base(),
                  v.end(),
                  something)),
        v.end());
```

Once you've got that, figuring out the *`something`*s isn't terribly difficult, and the next thing you know, you have the code in the original example. That's why this kind of statement is often known as "write-only" code. As you write the code, it seems straightforward, because it's a natural outgrowth of some basic ideas (e.g., the `erase-remove` idiom plus the notion of using `find` with reverse iterators). *Readers*, however, have great difficulty in decomposing the final product back into the ideas on which it is based. That's the calling card of write-only code: it's easy to write, but it's hard to read and understand.

Whether code is write-only depends on who's reading it. As I noted, some C++ programmers think nothing of the code in this Item. If that's typical in the environment in which you work and you expect it to be typical in the future, feel free to unleash your most advanced STL programming inclinations. However, if your colleagues are less comfortable with a functional programming style and are less experienced with the STL, scale back your ambitions and write something more along the lines of the two-statement alternative I showed earlier.

It's a software engineering truism that code is read more often than it is written. Equally well established is that software spends far more time in maintenance than it does in development. Software that cannot be read and understood cannot be maintained, and software that cannot be maintained is hardly worth having. The more you work with the STL, the more comfortable you'll become with it, and the more you'll feel the pull to nest function calls and create function objects on the fly. There's nothing wrong with that, but always bear in mind that the code you write today will be read by somebody —possibly you—someday in the future. Prepare for that day.

Use the STL, yes. Use it well. Use it effectively. But avoid producing write-only code. In the long run, such code is anything *but* effective.

Item 48: Always `#include` the proper headers

Among the minor frustrations of STL programming is that it is easy to create software that compiles on one platform, yet requires additional `#include` directives on others. This annoyance stems from the fact that the Standard for C++ (unlike the Standard for C) fails to dictate which standard headers must or may be `#include`d by other standard headers. Given such flexibility, different implementers have chosen to do different things.

To give you some idea of what this means in practice, I sat down one day with five STL platforms (let's call them A, B, C, D, and E), and I spent a little time throwing toy programs at them to see which standard headers I could omit and still get a successful compilation. This indirectly told me which headers `#include` other headers. This is what I found:

- With A and C, `<vector>` `#include`S `<string>`.
- With C, `<algorithm>` `#include`S `<string>`.
- With C and D, `<iostream>` `#include`S `<iterator>`.
- With D, `<iostream>` `#include`S `<string>` and `<vector>`.
- With D and E, `<string>` `#include`S `<algorithm>`.
- With all five implementations, `<set>` `#include`S `<functional>`.

Except for the case of `<set>` `#include`ing `<functional>`, I didn't find a way to get a program with a missing header past implementation B. According to Murphy's Law, then, you will always develop under a platform like A, C, D, or E and you will always be porting to a platform like B, especially when the pressure for the port is greatest and the time to accomplish it is least. Naturally.

But don't blame your compilers or library implementations for your porting woes. It's your fault if you're missing required headers. Any time you refer to elements of namespace `std`, you are responsible for having `#include`d the appropriate headers. If you omit them, your code might compile anyway, but you'll still be missing necessary headers, and other STL platforms may justly reject your code.

To help you remember what's required when, here's a quick summary of what's in each standard STL-related header:

- Almost all the containers are declared in headers of the same name, i.e., `vector` is declared in `<vector>`, `list` is declared in `<list>`, etc. The exceptions are `<set>` and `<map>`. `<set>` declares both `set` and `multiset`, and `<map>` declares both `map` and `multimap`.
- All but four algorithms are declared in `<algorithm>`. The exceptions are `accumulate` (see Item 37), `inner_product`, `adjacent_difference`, and `partial_sum`. Those algorithms are declared in `<numeric>`.
- Special kinds of iterators, including `istream_iterator`s and `istreambuf_iterators` (see Item 29), are declared in `<iterator>`.
- Standard functors (e.g., `less<T>`) and functor adapters (e.g., `not1`, `bind2nd`) are declared in `<functional>`.

Any time you use any of the components in a header, be sure to provide the corresponding `#include` directive, even if your development platform lets you get away without it. Your diligence will pay off in reduced stress when you find yourself porting to a different platform.

Item 49: Learn to decipher STL-related compiler diagnostics

It's perfectly legal to define a `vector` with a particular size,

```
vector<int> v(10);              // create a vector of size 10
```

and `string`s act a lot like `vector`s, so you might expect to be able to do this:

```
string s(10);                  // attempt to create a string
of size 10
```

This won't compile. There is no `string` constructor taking an `int` argument. One of my STL platforms tells me that like this:

```
example.cpp(20) : error C2664: '__thiscall
std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char>
>::std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> >(const class std::allocator<char> &)' :
cannot convert
parameter 1 from 'const int' to 'const class
std::allocator<char> &'

Reason: cannot convert from 'const int' to 'const class
std::allocator<char>'

No constructor could take the source type, or constructor
overload
resolution was ambiguous
```

Isn't that wonderful? The first part of the message looks as if a cat walked across the keyboard, the second part mysteriously refers to an allocator never mentioned in the source code, and the third part says the constructor call is bad. The third part is accurate, of course, but let's first focus our attention on the result of the purported feline stroll, because it's representative of diagnostics you'll frequently see when using `string`s.

`string` isn't a class, it's a `typedef`. In particular, it's a typedef for this:

```
basic_string<char, char_traits<char>, allocator<char> >
```

That's because the C++ notion of a string has been generalized to mean sequences of arbitrary character types with arbitrary character characteristics ("traits") and stored in memory allocated by arbitrary allocators. All `string`-like objects in C++ are really instantiations of the template `basic_string`, and that's why most compilers refer to the type `basic_string` when they issue diagnostics about programs making erroneous use of `string`s. (A few compilers are kind enough to use the name `string` in diagnostics, but most aren't.) Often, such diagnostics will explicitly note that `basic_string` (and the attendant helper templates `char_traits` and `allocator`) are in the `std` namespace, so it's not uncommon to see errors involving `string`s yield diagnostics that mention this type:

```
std::basic_string<char, std::char_traits<char>,
std::allocator<char> >
```

This is quite close to what's used in the compiler diagnostic above, but different compilers use variations on the theme. Another STL platform I use refers to `string`s this way,

```
basic_string<char,string_char_traits<char>,__default_alloc_te
>
```

The names `string_char_traits` and `__default_alloc_template` are nonstandard, but that's life. Some STL implementations deviate from the standard. If you don't like the deviations in your current STL implementation, consider replacing it with a different one. Item 50 gives examples of places you can go for alternative implementations.

Regardless of how a compiler diagnostic refers to the `string` type, the technique for reducing the diagnostic to something meaningful is the same: globally replace the `basic_string` gobbledegook with the text "`string`". If you're using a command-line compiler, it's usually easy to do this with a program like `sed` or a scripting language like `perl`, `python`, or `ruby`. (You'll find an example of such a script in Zolman's article, "An STL Error Message Decryptor for Visual C++" [26].) In the case of the diagnostic above, we globally replace

```
std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> >
```

with `string` and we end up with this:

```
example.cpp(20) : error C2664: '__thiscall
string::string(const class
std::allocator<char> &)' : cannot convert parameter 1 from
'const int' to
'const class std::allocator<char> &'
```

This makes clear (or at least clearer) that the problem is in the type of the parameter passed to the `string` constructor, and even though the mysterious reference to `allocator<char>` remains, it should be easy to look up the constructor forms for `string` to see that none exists taking only a size.

By the way, the reason for the mysterious reference to an allocator is that each standard container has a constructor taking only an allocator. In the case of `string`, it's one of three constructors that can be called with one argument, but for some reason, this compiler figures that the one taking an allocator is the one you're trying to call. The compiler figures wrong, and the diagnostic is misleading. Oh well.

As for the constructor taking only an allocator, please don't use it. That constructor makes it easy to end up with containers of the same type but with inequivalent allocators. In general, that's bad. Very bad. To find out why, turn to Item 11.

Now let's tackle a more challenging diagnostic. Suppose you're implementing an email program that allows users to refer to people by nicknames instead of by email addresses. For example, such a program would make it possible to use "The Big Cheese" as a synonym for the email address of the President of the United States (which happens to be president@whitehouse.gov). Such a program might use a map from nicknames to email addresses, and it might offer a member function `showEmailAddress` that displays the email address associated with a given nickname:

Click here to view code image

```
class NiftyEmailProgram {
private:
  typedef map<string, string> NicknameMap;

  NicknameMap nicknames;               // map from nicknames to
                                       // email addresses
public:
  ...
  void showEmailAddress(const string& nickname) const;
};
```

Inside `showEmailAddress`, you'll need to find the map entry associated with a particular nickname, so you might write this:

Click here to view code image

```
void NiftyEmailProgram::showEmailAddress(const string&
nickname) const
{
  ...
  NicknameMap::iterator i = nicknames.find(nickname);
  if (i != nicknames.end()) ...
  ...
}
```

Compilers don’t like this, and with good reason, but the reason isn’t obvious. To help you figure it out, here’s what one STL platform helpfully emits:

Click here to view code image

```
example.cpp(17) : error C2440: 'initializing' : cannot
convert from 'class std::_Tree<class
std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> >,struct
std::pair<class std::basic_string<char,struct
std::char_traits<char>,class
std::allocator<char> > const ,class
std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> > >,struct
std::map<class
std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> >,class
std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> >,struct
std::less<class std::basic_string<char,struct
std::char_traits<char>,class
std::allocator<char> > >,class std::allocator<class
std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> > >
>::_Kfn,struct std::less<class
std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> > >,class
std::allocator<class std::basic_string<char,struct
std::char_traits<char>,class
std::allocator<char> > > >::const_iterator' to 'class
std::_Tree<class
std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> >,struct
std::pair<class std::basic_string<char,struct
std::char_traits<char>,class
std::allocator<char> > const ,class
std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> > >,struct
std::map<class
std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> >,class
std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> >,struct
std::less<class std::basic_string<char,struct
std::char_traits<char>,class
std::allocator<char> > >,class std::allocator<class
```

```
std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> > >
>::_Kfn,struct std::less<class
std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> > >,class
std::allocator<class std::basic_string<char,struct
std::char_traits<char>,class
std::allocator<char> > > >::iterator'

No constructor could take the source type, or constructor
overload resolution was
ambiguous
```

At 2095 characters long, this message looks fairly gruesome, but I’ve seen worse. One of my favorite STL platforms produces a diagnostic of 4812 characters for this example. As you might guess, features other than its error messages are what have engendered my fondness for it.

Let’s reduce this mess to something manageable. We begin with the replacement of the `basic_string` gibberish with `string`. That yields this:

Click here to view code image

```
example.cpp(17) : error C2440: 'initializing' : cannot
convert from 'class
std::_Tree<class string,struct std::pair<class string const
,class string
>,struct std::map<class string,class string,struct
std::less<class string
>,class std::allocator<class string > >::_Kfn,struct
std::less<class string
>,class std::allocator<class string > >::const_iterator' to
'class
std::_Tree<class string,struct std::pair<class string const
,class string
>,struct std::map<class string,class string,struct
std::less<class string
>,class std::allocator<class string > >::_Kfn,struct
std::less<class string
>,class std::allocator<class string > >::iterator'

No constructor could take the source type, or constructor
overload
resolution was ambiguous
```

Much better. Now a svelte 745 characters long, we can start to actually look at the message. One of the things that is likely to catch our eye is the mention of the template `std::_Tree`. The Standard says nothing about a template called `_Tree`, but the leading underscore in the name followed by a capital letter jogs our memory that such names are reserved for implementers. This is an internal template used to implement some part of the STL.

In fact, almost all STL implementations use some kind of underlying template to implement the standard associative containers (`set`, `multiset`, `map`, and `multimap`). In the same way that source code using `string` typically leads to diagnostics mentioning `basic_string`, source code using a standard associative container often leads to diagnostics mentioning some underlying tree template. In this case, it's called `_Tree`, but other implementations I know use `__tree` or `__rb_tree`, the latter reflecting the use of red-black trees, the most common type of balanced tree used in STL implementations.

Setting `_Tree` aside for a moment, the message above mentions a type we should recognize: `std::map<class string,class string,struct std::less<class string >,class std::allocator<class string > >`. This is precisely the type of `map` we are using, except that the comparison and allocator types (which we chose not to specify when we defined the map) are shown. The error message will be easier to understand if we replace that type with our typedef for it, `NicknameMap`. That leads to this:

```
example.cpp(17) : error C2440: 'initializing' : cannot
convert from 'class
std::_Tree<class string,struct std::pair<class string
const,class string
>,struct NicknameMap::_Kfn,struct std::less<class string
>,class
std::allocator<class string > >::const_iterator' to 'class
std::_Tree<class
string,struct std::pair<class string const ,class string
>,struct
NicknameMap::_Kfn,struct std::less<class string >,class
std::allocator<class
string > >::iterator'

No constructor could take the source type, or constructor
overload
resolution was ambiguous
```

This message is shorter, but not much clearer. We need to do something with `_Tree`. Because `_Tree` is an implementation-specific template, the only way to know the meaning of its template parameters is to read the source code, and there's no reason to go rummaging through implementation-specific source code if we don't have to. Let's try simply replacing all the stuff passed to `_Tree` with `SOMETHING` to see what we get. This is the result:

```
example.cpp(17) : error C2440: 'initializing' : cannot
convert from 'class
std::_Tree<SOMETHING>::const_iterator' to 'class
std::_Tree<SOMETHING>::iterator'

No constructor could take the source type, or constructor
overload
resolution was ambiguous
```

This is something we can work with. The compiler is complaining that we're trying to convert some kind of `const_iterator` into an `iterator`, a clear violation of `const` correctness. Let's look again at the offending code, where I've highlighted the line raising the compiler's ire:

Click here to view code image

```
class NiftyEmailProgram {
private:
  typedef map<string, string> NicknameMap;

  NicknameMap nicknames;

public:
  ...
  void showEmailAddress(const string& nickname) const;
};

void NiftyEmailProgram::showEmailAddress(const string&
nickname) const
{
  ...
  NicknameMap::iterator i = nicknames.find(nickname);
  if (i != nicknames.end()) ...
  ...
}
```

The only interpretation that makes any sense is that we're trying to initialize `i` (which is an `iterator`) with a `const_iterator` returned from `map::find`. That seems odd, because we're calling `find` on `nicknames`, and `nicknames` is a non-`const` object. `find` should thus return a non-`const` iterator.

Look again. Yes, `nicknames` is declared as a non-`const map`, but `showEmailAddress` is a *`const`* member function, and inside a `const` member function, all non-static data members of the class become `const`! Inside `showEmailAddress`, `nicknames` *is* a `const map`. Suddenly the error message makes sense. We're trying to generate an `iterator` into a map we've promised not to modify. To fix the problem, we must either make `i` a `const_iterator` or we must make `showEmailAddress` a non-`const` member function. Both solutions are probably less challenging than ferreting out the meaning of the error message.

In this Item, I've shown textual substitutions to reduce the complexity of error messages, but once you've practiced a little, you'll be able to perform the substitutions in your head most of the time. I'm no musician (I have trouble turning on the radio), but I'm told that good musicians can sight-read several bars at a glance; they don't need to look at individual notes. Experienced STL programmers develop a similar skill. They can internally translate things like `std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >` into `string` without thinking about it. You, too, will develop this skill, but until you do, remember that you can almost always reduce compiler diagnostics to something comprehensible by replacing lengthy template-based type names with shorter mnemonics. In many cases, all you have to do is replace typedef expansions with typedef names you're already using. That's what we did when we replaced `std::map<class string,class string,struct std::less<class string >,class std::allocator<class string > >` with `NicknameMap`.

Here are a few other hints that should help you make sense of STL-related compiler messages:

- For `vector` and `string`, iterators are sometimes pointers, so compiler diagnostics may refer to pointer types if you've made a mistake with an `iterator`. For example, if your source code refers to `vector<double>::iterator`s, compiler messages will sometimes mention `double*` pointers. (A noteworthy exception is when you're using the STL implementation from STLport and you're running in debug mode. In that case, `vector` and `string` iterators are definitely not pointers. For more on STLport and its debug mode, turn to Item 50.)
- Messages mentioning `back_insert_iterator`, `front_insert_iterator`, or `insert_iterator` almost always mean you've made a mistake calling `back_inserter`, `front_inserter`, or `inserter`, respectively. (`back_inserter`

returns an object of type `back_insert_iterator`, `front_inserter` returns an object of type `front_insert_iterator`, and `inserter` returns an object of type `insert_iterator`. For information on the use of these inserters, consult Item 30.) If you didn't call these functions, some function you called (directly or indirectly) did.

- Similarly, if you get a message mentioning `binder1st` or `binder2nd`, you've probably made a mistake using `bind1st` or `bind2nd`. (`bind1st` returns an object of type `binder1st`, and `bind2nd` returns an object of type `binder2nd`.)
- Output iterators (e.g., `ostream_iterator`s, `ostreambuf_iterator`s (see Item 29), and the iterators returned from `back_inserter`, `front_inserter`, and `inserter`) do their outputting or inserting work inside assignment operators, so if you've made a mistake with one of these iterator types, you're likely to get a message complaining about something inside an assignment operator you've never heard of. To see what I mean, try compiling this code:

Click here to view code image

```
vector<string*> v;                              // try to
print a container
copy(v.begin(), v.end(),                        // of
string* pointers as
     ostream_iterator<string>(cout, "\n"));     // string
objects
```

- If you get an error message originating from inside the implementation of an STL algorithm (i.e., the source code giving rise to the error is in `<algorithm>`), there's probably something wrong with the types you're trying to use with that algorithm. For example, you may be passing iterators of the wrong category. To see how such usage errors are reported, edify (and amuse!) yourself by feeding this to your compilers:

Click here to view code image

```
list<int>::iterator i1, i2;                     // pass
bidirectional iterators to
sort(i1, i2);                                   // an
algorithm requiring random
                                                // access
iterators
```

- If you're using a common STL component like `vector`, `string`, or the `for_each` algorithm, and a compiler says it has no idea what you're talking about, you've probably failed to `#include` a required header file. As Item 48 explains, this problem can befall code that has been compiling smoothly for quite some time if you port it to a new platform.

Item 50: Familiarize yourself with STL-related web sites

The Internet is rife with STL information. Ask your favorite search engine to look for "STL", and it's sure to return hundreds of links, some of which may actually be relevant. For most STL programmers, however, no searching is necessary. The following sites are likely to rise to the top of almost everybody's most-frequently-used list:

- **The SGI STL site**, http://www.sgi.com/tech/stl/.
- **The STLport site**, http://www.stlport.org/.
- **The Boost site**, http://www.boost.org/.

What follows are brief descriptions of why these sites are worth bookmarking.

The SGI STL Web Site

SGI's STL web site tops the list, and for good reason. It offers comprehensive documentation on every component of the STL. For many programmers, this site is their on-line reference manual, regardless of which STL platform they are using. (The reference documentation was put together by Matt Austern, who later extended and polished it for his *Generic Programming and the STL* [4].) The material here covers more than just the STL components themselves. *Effective STL*'s discussion of thread safety in STL containers (see Item 12), for example, is based on the treatment of the topic at the SGI STL web site.

The SGI site offers something else for STL programmers: a freely downloadable implementation of the STL. This implementation has been ported to only a handful of compilers, but the SGI distribution is also the basis for the widely ported STLport distribution, about which I write more in a moment. Furthermore, the SGI implementation of the STL offers a number of nonstandard components that can make STL programming even more powerful, flexible, and fun. Foremost among these are the following:

- The **hashed associative containers** `hash_set`, `hash_multiset`, `hash_map`, and `hash_multimap`. For more information about these containers, turn to Item 25.
- A **singly linked list container**, `slist`. This is implemented as you'd imagine, and iterators point to the list nodes you'd expect them to point to. Unfortunately, this makes it expensive to implement the `insert` and `erase` member functions, because both require adjustment of the `next` pointer of the node *preceding* the node pointed to by the iterator. In a doubly linked list (such as the standard `list` container), this isn't a problem, but in a singly linked list, going "back" one node is a linear-time operation. For SGI's `slist`, `insert` and `erase` take linear instead of constant time, a considerable drawback. SGI addresses the problem through the nonstandard (but constant-time) member functions `insert_after` and `erase_after`. Notes SGI,

> If you find that `insert_after` and `erase_after` aren't adequate for your needs and that you often need to use `insert` and `erase` in the middle of the list, you should probably use `list` instead of `slist`.

Dinkumware also offers a singly linked list container called `slist`, but it uses a different iterator implementation that preserves the constant-time performance of `insert` and `erase`. For more information on Dinkumware, consult Appendix B.

- A **`string`-like container for very large strings**. The container is called `rope`, because a rope is a heavy-duty string, don't you see? SGI describes `rope`s this way:

> Ropes are a scalable string implementation: they are designed for efficient operations that involve the string as a whole. Operations such as assignment, concatenation, and substring take time that is nearly independent of the length of the string. Unlike C strings, ropes are a reasonable representation for very long strings, such as edit buffers or mail messages.

Under the hood, `rope`s are implemented as trees of reference-counted substrings, and each substring is stored as a `char` array. One interesting aspect of the `rope` interface is that the `begin` and `end` member functions always return `const_iterator`s. This is to discourage clients from performing operations that change individual characters. Such operations are expensive. `rope`s are optimized for actions that involve entire strings (e.g., assignment, concatenation, and taking substrings, as mentioned above); single-character operations perform poorly.

- A variety of **nonstandard function objects and adapters**. The original HP STL implementation included more functor classes than made it into standard C++. Two of the more widely missed by old-time STL hackers are `select1st` and `select2nd`, because they are so useful for working with `map`s and `multimap`s. Given a `pair`, `select1st` returns its first component and `select2nd` returns its second. These nonstandard functor class templates can be used as follows:

Click here to view code image

```
map<int, string> m;

...

// write all the map keys to cout
transform(m.begin(), m.end(),
          ostream_iterator<int>(cout, "\n"),
          select1st<map<int, string>::value_type>());

// create a vector and copy all the values in the map into it
vector<string> v;
transform(m.begin(), m.end(), back_inserter(v),
          select2nd<map<int, string>::value_type>());
```

As you can see, `select1st` and `select2nd` make it easy to use algorithm calls in places where you might otherwise have to write your own loops (see Item 43), but, if you use these functors, the fact that they are nonstandard leaves you open to the charge that you are writing unportable and unmaintainable code (see Item 47). Diehard STL aficionados don’t care. They consider it an injustice that `select1st` and `select2nd` didn’t make it into the Standard in the first place.

Other nonstandard function objects that are part of the SGI implementation include `identity`, `project1st`, `project2nd`, `compose1` and `compose2`. To find out what these do, you’ll have to visit the web site, though you’ll find an example use of `compose2` on page 187 of this book.

> By now, I hope it’s clear that visiting the SGI web site will certainly be rewarding.

SGI’s library implementation goes beyond the STL. Their goal is the development of a complete implementation of the standard C++ library, except for the parts inherited from C. (SGI assumes you already have a standard C library at your disposal.) As a result, another noteworthy download available from SGI is an implementation of the C++ iostreams library. As you might expect, this implementation integrates well with SGI’s implementation of the STL, but it also features performance that’s superior to that of many iostream implementations that ship with C++ compilers.

The STLport Web Site

STLport’s primary selling point is that it offers a modified version of SGI’s STL implementation (including iostreams, etc.) that’s been ported to more than 20 compilers. Like SGI’s library, STLport’s is available for free download. If you’re writing code that has to work on multiple platforms, you may be able to save yourself a wheelbarrow of grief by standardizing on the STLport implementation and using it with all your compilers.

Most of STLport’s modifications to SGI’s code base focus on improved portability, but STLport’s STL is also the only implementation I know that offers a “debug mode” to help detect improper use of the STL—uses that compile but lead to undefined runtime behavior. For example, Item 30 uses this example in its discussion of the common mistake of writing beyond the end of a container:

Click here to view code image

```
int transmogrify(int x);                        // this
function produces
                                                // some new
value from x

vector<int> values;

...                                             // put data
into values

vector<int> results;
transform(values.begin(), values.end(),         // this will
attempt to
```

```
          results.end(),                       // write
beyond the
          transmogrify);                       // end of
results!
```

This will compile, but when run, it yields undefined results. If you're lucky, something horrible will happen inside the call to `transform`, and debugging the problem will be relatively straightforward. If you're not lucky, the call to `transform` will trash data somewhere in your address space, but you won't discover that until later. At that point, determining the cause of the memory corruption will be—shall we say?—*challenging*.

STLport's debug mode all but eliminates the challenge. When the above call to `transform` is executed, the following message is generated (assuming STLport is installed in the directory `C:\STLport`):

```
C:\STLport\stlport\stl\debug\_iterator.h:265 STL assertion
failure :
_Dereferenceable(*this)
```

The program then stops, because STLport debug mode calls `abort` if it encounters a usage error. If you'd prefer to have an exception thrown instead, you can configure STLport to do things your way.

Admittedly, the above error message isn't as clear as it might be, and it's unfortunate that the reported file and line correspond to the location of the internal STL assertion instead of the line calling `transform`, but this is still a lot better than running past the call to `transform`, then trying to figure out why your data structures are corrupt. With STLport's debug mode, all you need to do is fire up your debugger and walk the call stack back into the code you wrote, then determine what you did wrong. Finding the offending source line is generally not a problem.

STLport's debug mode detects a variety of common errors, including passing invalid ranges to algorithms, attempting to read from an empty container, using an iterator from one container as the argument to a second container's member function, etc. It accomplishes this magic by having iterators and their containers track one another. Given two iterators, it's thus possible to check to see if they come from the same container, and when a container is modified, it's possible to invalidate the appropriate set of iterators.

Because STLport uses special iterator implementations in debug mode, iterators for `vector` and `string` are class objects instead of raw pointers. Hence, using STLport and compiling in debug mode is a good way to make sure that nobody is getting sloppy about the difference between pointers and iterators for these container types. That alone may be reason enough to give STLport's debug mode a try.

The Boost Web Site

In 1997, when the closing bell rang on the process that led to the International Standard for C++, some people were disappointed that library features they'd advocated hadn't made the cut. Some of these people were members of the Committee itself, so they set out to lay the foundation for additions to the standard library during the second round of standardization. The result is Boost, a web site whose mission is to "provide free, peer-reviewed, C++ libraries. The emphasis is on portable libraries which work well with the C++ Standard Library." Behind the mission is a motive:

> To the extent a library becomes "existing practice", the likelihood increases that someone will propose it for future standardization. Submitting a library to Boost.org is one way to establish existing practice... .

In other words, Boost offers itself as a vetting mechanism to help separate the sheep from the goats when it comes to potential additions to the standard C++ library. This is a worthy service, and we should all be grateful.

Another reason to be grateful is the collection of libraries you'll find at Boost. I won't attempt to describe them all here, not least because new ones will doubtless have been added by the time you read these words. For STL users, however, two kinds of libraries are particularly relevant. The first is the smart pointer library featuring `shared_ptr`, the template for reference-counted smart pointers that, unlike the standard library's `auto_ptr`, may safely be stored in STL containers (see Item 8). Boost's smart pointer library also offers `shared_array`, a reference-counted smart pointer for dynamically allocated arrays, but Item 13 argues that dynamically allocated arrays are inferior to `vector`s and `string`s, and I hope you find its argument persuasive.

Boost's second attraction for STL fans is its bevy of STL-related function objects and associated facilities. These libraries comprise a fundamental redesign and reimplementation of the ideas behind STL function objects and adapters, and the results eliminate a host of restrictions that artificially limit the utility of the standard functors. As an example of such a restriction, you’ll find that if you try to use `bind2nd` with `mem_fun` or `mem_fun_ref` (see Item 41) to bind an object to a member function’s parameter and that member function takes its parameter by reference, your code is unlikely to compile. You’ll find the same if you try to use `not1` or `not2` with `ptr_fun` and a function declaring a by-reference parameter. The cause in both cases is that during template instantiation, most STL platforms generate a reference to a reference, and references to references are not legal in C++. (The Standardization Committee is mulling over a change in the Standard to address this matter.) Here’s an example of what has become known as “the reference-to-reference problem:”

Click here to view code image

```
class Widget {
public:
...
int readStream(istream& stream);                 //
readStream takes
...                                              // its
parameter by
};                                                //
reference

vector<Widget*> vw;

...

for_each(                                         // most STL
platforms
  vw.begin(), vw.end(),                           // try to
generate a
  bind2nd(mem_fun(&Widget::readStream), cin)      //
reference to a
);                                                //
reference in this
                                                  // call;
such code
                                                  // won't
compile
```

Boost’s function objects avoid this and other issues, plus they expand the expressiveness of function objects considerably.

If you’re intrigued by the potential of STL function objects and you want to explore it further, hurry over to Boost right away. If you abhor function objects and think they exist only to pacify a vocal minority of Lisp apologists turned C++ programmers, hurry over to Boost anyway. Boost’s function object libraries are important, but they make up only a small part of what you’ll find at the site.

Bibliography

Most of the publications below are cited in this book, though many citations occur only in the Acknowledgments. Uncited publications are bulleted instead of having a number.

Given the instability of Internet URLs, I hesitated before including them in this bibliography. In the end, I decided that even if a URL is broken by the time you try it, knowing where a document used to reside should help you find it at a different URL.

Things I Wrote

[1] Scott Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs* (second edition), Addison-Wesley, 1998, ISBN 0-201-92488-9. Also available on *Effective C++ CD* (see below).

- Scott Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (third edition), Addison-Wesley, 2005, ISBN 0-321-33487-6.

[2] Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996, ISBN 0-201-63371-X. Also available on *Effective C++ CD* (see below).

- Scott Meyers, *Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, 1999, ISBN 0-201-31015-5. Contains *Effective C++* (second edition), *More Effective C++*, and several related magazine articles.

Things I Didn’t Write (But Wish I Had)

[3] Nicolai M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, 1999, ISBN 0-201-37926-0. An indispensable book. Every C++ programmer should have ready access to a copy.

[4] Matthew H. Austern, *Generic Programming and the STL*, Addison-Wesley, 1999, ISBN 0-201-30956-4. This book is essentially a printed version of the material at the SGI STL Web Site,

http://www.sgi.com/tech/stl/.

[5] ISO/IEC, *International Standard, Programming Languages—C++* (second edition), Reference Number ISO/IEC 14882:1998(E), 1998. The official document describing C++. Available in PDF from ANSI at http://webstore.ansi.org/ansidocstore/default.asp.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995, ISBN 0-201-63361-2. Also available as *Design Patterns CD*, Addison-Wesley, 1998, ISBN 0-201-63498-8. The definitive book on design patterns. Every practicing C++ programmer should be familiar with the patterns described here and should have easy access to the book or the CD.

[7] Bjarne Stroustrup, *The C++ Programming Language* (third edition), Addison-Wesley, 1997, ISBN 0-201-88954-4. The "resource acquisition is initialization" idiom I mention in Item 12 is discussed in section 14.4.1 of this book, and the code I refer to in Item 36 is on page 530.

[8] Herb Sutter, *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*, Addison-Wesley, 2000, ISBN 0-201-61562-2. An exemplary complement to my *Effective* books, I'd laud this even if Herb hadn't asked me to write the Foreword for it.

[9] Herb Sutter, *More Exceptional C++: 40 More Engineering Puzzles, Programming Problems, and Solutions*, Addison-Wesley, scheduled for publication in 2001, tentative ISBN 0-201-70434-X. Based on the draft I've seen, this looks to be every bit as good as its predecessor [8].

[10] Dov Bulka and David Mayhew, *Efficient C++: Performance Programming Techniques*, Addison-Wesley, 2000, ISBN 0-201-37950-3. The only book devoted to efficiency in C++, hence the best.

[11] Matt Austern, "How to Do Case-Insensitive String Comparison," *C++ Report*, May 2000. This article is so useful, it's reproduced as Appendix A of this book.

[12] Herb Sutter, "When Is a Container Not a Container?," *C++ Report*, May 1999. Available at http://www.gotw.ca/publications/mill09.htm. Revised and updated as Item 6 in *More Exceptional C++* [9].

[13] Herb Sutter, "Standard Library News: `set`s and `map`s," *C++ Report*,

October 1999. Available at http://www.gotw.ca/publications/mill11.htm. Revised and updated as Item 8 in *More Exceptional C++* [9].

[14] Nicolai M. Josuttis, "Predicates vs. Function Objects," *C++ Report*, June 2000.

[15] Matt Austern, "Why You Shouldn't Use `set`—and What to Use Instead," *C++ Report*, April 2000.

[16] P. J. Plauger, "Hash Tables," *C/C++ Users Journal*, November 1998. Describes the Dinkumware approach to hashed containers (discussed in Item 25) and how it differs from competing designs.

[17] Jack Reeves, "STL Gotcha's," *C++ Report*, January 1997. Available at http://www.bleading-edge.com/Publications/C++Report/v9701/abstract.htm.

[18] Jack Reeves, "Using Standard `string` in the Real World, Part 2," *C++ Report*, January 1999. Available at http://www.bleading-edge.com/Publications/C++Report/v9901/abstract.htm.

[19] Andrei Alexandrescu, "Traits: The `else-if-then` of Types," *C++ Report*, April 2000. Available at http://erdani.org/publications/traits.html.

[20] Herb Sutter, "Optimizations That Aren't (In a Multithreaded World)," *C/C++ Users Journal*, June 1999. Available at http://www.gotw.ca/publications/optimizations.htm. Revised and updated as Item 16 in *More Exceptional C++* [9].

[21] The SGI STL web site, http://www.sgi.com/tech/stl/. Item 50 summarizes the material at this important site. The page on thread safety in STL containers (which motivated Item 12) is at http://www.sgi.com/tech/stl/thread_safety.html.

[22] The Boost web site, http://www.boost.org/. Item 50 summarizes the material at this important site.

[23] Nicolai M. Josuttis, "User-Defined Allocator," http://www.josuttis.com/cppcode/allocator.html. This page is part of the web site for Josuttis' excellent book on C++'s standard library [3].

[24] Matt Austern, "The Standard Librarian: What Are Allocators Good For?," *C/C++ Users Journal*'s C++ Experts Forum (an on-line

extension to the magazine), December 2000, http://www.cuj.com/documents/s=8000/cujcexp1812austern/. Good information on allocators is hard to come by. This column complements the material in Items 10 and 11. It also includes a sample allocator implementation.

[25] Klaus Kreft and Angelika Langer, “A Sophisticated Implementation of User-Defined Inserters and Extractors,” *C++ Report*, February 2000.

[26] Leor Zolman, “An STL Error Message Decryptor for Visual C++,” *C/C++ Users Journal*, July 2001. This article and the software it describes are available at http://www.bdsoft.com/tools/stlfilt.html.

[27] Bjarne Stroustrup, “Sixteen Ways to Stack a Cat,” *C++ Report*, October 1990. Available at http://www.research.att.com/~bs/stack_cat.pdf.

- Herb Sutter, “Guru of the Week #74: Uses and Abuses of `vector`,” September 2000, http://www.gotw.ca/gotw/074.htm. This quiz (and accompanying solution) does a good job of making you think about such `vector`-related issues as size vs. capacity (see Item 14), but it also discusses why algorithm calls are often superior to hand-written loops (see Item 43).
- Matt Austern, “The Standard Librarian: Bitsets and Bit Vectors,” *C/C++ Users Journal*’s C++ Experts Forum (an on-line extension to the magazine), May 2001, http://www.ddj.com/cpp/184401382. This article provides information on `bitset`s and how they compare to `vector<bool>`, topics I briefly examine in Item 18.

Things I Had To Write (But Wish I Hadn’t)

- The *Effective C++* Errata List, http://www.aristeia.com/BookErrata/ec++3e-errata.html.

[28] The *More Effective C++* Errata List, http://www.aristeia.com/BookErrata/mec++-errata.html.

- The *Effective C++ CD* Errata List, http://www.aristeia.com/BookErrata/cd1e-errata.html.

[29] The *More Effective C++* `auto_ptr` Update page,

http://www.aristeia.com/BookErrata/auto_ptr-update.html.

Appendix A. Locales and Case-Insensitive String Comparisons

Item 35 explains how to use `mismatch` and `lexicographical_compare` to implement case-insensitive string comparisons, but it also points out that a truly general solution to the problem must take locales into account. This book is about the STL, not internationalization, so I have next to nothing to say about locales. However, Matt Austern, author of *Generic Programming and the STL* [4], addressed the locale aspects of case-insensitive string comparison in a column in the May 2000 *C++ Report* [11]. In the interest of telling the whole story on this important topic, I am pleased to reprint his column here, and I am grateful to Matt and to 101communications for granting me permission to do so.

How to Do Case-Insensitive String Comparison by Matt Austern

If you've ever written a program that uses strings (and who hasn't?), chances are you've sometimes needed to treat two strings that differ only by case as if they were identical. That is, you've needed comparisons—equality, less than, substring matches, sorting—to disregard case. And, indeed, one of the most frequently asked questions about the Standard C++ Library is how to make strings case insensitive. This question has been answered many times. Many of the answers are wrong.

First of all, let's dispose of the idea that you should be looking for a way to write a case-insensitive string class. Yes, it's technically possible, more or less. The Standard Library type `std::string` is really just an alias for the template `std::basic_string<char, std::char_traits<char>, std::allocator<char> >`. It uses the traits parameter for all comparisons, so, by providing a traits parameter with equality and less than redefined appropriately, you can instantiate `basic_string` in such a way so that the `<` and `==` operators are case insensitive. You can do it, but it isn't worth the trouble.

- *You won't be able to do I/O*, at least not without a lot of pain. The I/O classes in the Standard Library, like `std::basic_istream` and `std::basic_ostream`, are templatized on character type and traits just like `std::basic_string` is. (Again, `std::ostream` is just an alias for `std::basic_ostream<char, char_traits<char> >`.) The traits parameters have to match. If you're using `std::basic_string<char, my_traits_class>` for your strings, you'll have to use `std::basic_ostream<char, my_traits_class>` for your stream output. You won't be able to use ordinary stream objects like `cin` and `cout`.

- *Case insensitivity isn't about an object, it's about how you use an object*. You might very well need to treat a string as case-sensitive in some contexts and case-insensitive in others. (Perhaps depending on a user-controlled option.) Defining separate types for those two uses puts an artificial barrier between them.
- *It doesn't quite fit*. Like all traits classes,[†] `char_traits` is small, simple, and stateless. As we'll see later in this column, proper case-insensitive comparisons are none of those things.
- *It isn't enough*. Even if all of `basic_string`'s own member functions were case insensitive, that still wouldn't help when you need to use nonmember generic algorithms like `std::search` and `std::find_end`. It also wouldn't help if you decided, for reasons of efficiency, to change from a container of `basic_string` objects to a string table.

A better solution, one that fits more naturally into the design of the Standard Library, is to ask for case-insensitive comparison when that's what you need. Don't bother with member functions like `string::find_first_of` and `string::rfind`; all of them duplicate functionality that's already there in nonmember generic algorithms. The generic algorithms, meanwhile, are flexible enough to accommodate case-insensitive strings. If you need to sort a collection of strings in case-insensitive order, for example, all you have to do is provide the appropriate comparison function object:

```
std::sort(C.begin(), C.end(), compare_without_case);
```

The remainder of this column will be about how to write that function object.

A first attempt

There's more than one way to alphabetize words. The next time you're in a bookstore, check how the authors' names are arranged: does Mary McCarthy come before Bernard Malamud, or after? (It's a matter of convention, and

I’ve seen it done both ways.) The simplest kind of string comparison, though, is the one we all learned in elementary school: lexicographic or “dictionary order” comparison, where we build up string comparison from character-by-character comparison.

Lexicographic comparison may not be suitable for specialized applications (no single method is; a library might well sort personal names and place names differently), but it’s suitable much of the time and it’s what string comparison means in C++ by default. Strings are sequences of characters, and if `x` and `y` are of type `std::string`, the expression `x < y` is equivalent to the expression

```
std::lexicographical_compare(x.begin(), x.end(), y.begin(),
y.end()).
```

In this expression, `lexicographical_compare` compares individual characters using `operator<`, but there’s also a version of `lexicographical_compare` that lets you choose your own method of comparing characters. That other version takes five arguments, not four; the last argument is a function object, a Binary Predicate that determines which of two characters should precede the other. All we need in order to use `lexicographical_compare` for case-insensitive string comparison, then, is to combine it with a function object that compares characters without regard to case.

The general idea behind case-insensitive comparison of characters is to convert both characters to upper-case and compare the results. Here’s the obvious translation of that idea into a C++ function object, using a well known function from the standard C library:

Click here to view code image

```
struct lt_nocase
  : public std::binary_function<char, char, bool> {
   bool operator()(char x, char y) const {
    return std::toupper(static_cast<unsigned char>(x)) <
           std::toupper(static_cast<unsigned char>(y));
   }
};
```

“For every complex problem there is a solution that is simple, neat, and wrong.” People who write books about C++ are fond of this class, because it makes a nice, simple example. I’m as guilty as anyone else; I use it in my book a half dozen times. It’s almost right, but that’s not good enough. The problem is subtle.

Here’s one example where you can begin to see the problem:

[Click here to view code image]

```
int main()
{
  const char* s1 = "GEW\334RZTRAMINER";
  const char* s2 = "gew\374rztraminer";
  printf("s1 = %s, s2 = %s\n", s1, s2);

  printf("s1 < s2: %s\n",
         std::lexicographical_compare(s1, s1 + 14, s2, s2 +
14, lt_nocase())
         ? "true" : "false");
}
```

You should try this out on your system. On my system (a Silicon Graphics O2 running IRIX 6.5), here’s the output:

```
s1 = GEWÜRZTRAMINER, s2 = gewürztraminer
s1 < s2: true
```

Hm, how odd. If you’re doing a case-insensitive comparison, shouldn’t `"gewürztraminer"` and `"GEWÜRZTRAMINER"` be the same? And now a slight variation: if you insert the line

```
setlocale(LC_ALL, "de");
```

just before the `printf` statement, suddenly the output changes:

```
s1 = GEWÜRZTRAMINER, s2 = gewürztraminer
s1 < s2: false
```

Case-insensitive string comparison is more complicated than it looks. This seemingly innocent program depends crucially on something most of us would prefer to ignore: locales.

Locales

A `char` is really nothing more than a small integer. We can choose to interpret a small integer as a character, but there’s nothing universal about that interpretation. Should some particular number be interpreted as a letter, a punctuation mark, or a nonprinting control character?

There’s no single right answer, and it doesn’t even make a difference as far as the core C and C++ languages are concerned. A few library functions need to make those distinctions: `isalpha`, for example, which determines whether a character is a letter, and `toupper`, which converts lowercase letters to uppercase and does nothing to uppercase letters or to characters that aren’t letters. All of that depends on local cultural and linguistic conventions: distinguishing between letters and nonletters means one thing in English, another thing in Swedish. Conversion from lower to upper case means something different in the Roman alphabet than in the Cyrillic, and means nothing at all in Hebrew.

By default the character manipulation functions work with a character set that’s appropriate for simple English text. The character `'\374'` isn’t affected by `toupper` because it isn’t a letter; it may look like `ü` when it’s printed on some systems, but that’s irrelevant for a C library routine that’s operating on English text. There is no `ü` character in the ASCII character set. The line

```
setlocale(LC_ALL, "de");
```

tells the C library to start operating in accordance with German conventions. (At least it does on IRIX. Locale names are not standardized.) There is a character `ü` in German, so `toupper` changes `ü` to `Ü`.

If this doesn’t make you nervous, it should. While `toupper` may look like a simple function that takes one argument, it also depends on a global variable —worse, a hidden global variable. This causes all of the usual difficulties: a function that uses `toupper` potentially depends on every other function in the entire program.

This can be disastrous if you’re using `toupper` for case-insensitive string comparison. What happens if you’ve got an algorithm that depends on a `list` being sorted (`binary_search`, say), and then a new locale causes the sort order to change out from under it? Code like this isn’t reusable; it’s barely usable. You can’t use it in libraries—libraries get used in all sorts of programs, not just programs that never call `setlocale`. You might be able to get away with using it in a large program, but you’ll have a maintenance problem: maybe you can prove that no other module ever calls `setlocale`, but can you prove that no other module in next year’s version of the program will call `setlocale`?

There’s no good solution to this problem in C. The C library has a single global locale, and that’s that. There is a solution in C++.

Locales in C++

A locale in the C++ Standard Library isn't global data buried deep within the library implementation. It's an object of type `std::locale`, and you can create it and pass it to functions just like any other object. You can create a locale object that represents the usual locale, for example, by writing

```
std::locale L = std::locale::classic();,
```

or you can create a German locale by writing

```
std::locale L("de");.
```

(As in the C library, names of locales aren't standardized. Check your implementation's documentation to find out what named locales are available.)

Locales in C++ are divided into *facets*, each of which handles a different aspect of internationalization, and the function `std::use_facet` extracts a specific facet from a locale object.[†] The `ctype` facet handles character classification, including case conversion. Finally, then, if `c1` and `c2` are of type `char`, this fragment will compare them in a case-insensitive manner that's appropriate to the locale `L`.

```
const std::ctype<char>& ct = std::use_facet<std::ctype<char>
>(L);
bool result = ct.toupper(c1) < ct.toupper(c2);
```

There's also a special abbreviation: you can write

```
std::toupper(c, L),
```

which (if `c` is of type `char`) means the same thing as

```
std::use_facet<std::ctype<char> >(L).toupper(c).
```

It's worth minimizing the number of times you call `use_facet`, though, because it can be fairly expensive.

Just as lexicographical comparison isn't appropriate for all applications, so character-by-character case conversion isn't always appropriate. (In German, for example, the lowercase letter "ß" corresponds to the uppercase sequence

---

[†] Warning: `use_facet` is a function template whose template parameter appears only in the return type, not in any of the arguments. Calling it uses a language feature called *explicit template argument specification*, and some C++ compilers don't support that feature yet. If you're using a compiler that doesn't support it, your library implementor may have provided a workaround so that you can call `use_facet` some other way.

“SS”.) Unfortunately, however, character-by-character case conversion is all we’ve got. Neither the C nor the C++ standard library provides any form of case conversion that works with anything more than one character at a time. If this restriction is unacceptable for your purposes, then you’re outside the scope of the standard library.

A digression: another facet

If you’re already familiar with locales in C++, you may have thought of another way to perform string comparisons: The `collate` facet exists precisely to encapsulate details of sorting, and it has a member function with an interface much like that of the C library function `strcmp`. There’s even a little convenience feature: if `L` is a locale object, you can compare two strings by writing `L(x, y)` instead of going through the nuisance of calling `use_facet` and then invoking a `collate` member function.

The “classic” locale has a `collate` facet that performs lexicographical comparison, just like `string`’s `operator<` does, but other locales perform whatever kind of comparison is appropriate. If your system happens to come with a locale that performs case-insensitive comparisons for whatever languages you’re interested in, you can just use it. That locale might even do something more intelligent than character-by-character comparisons!

Unfortunately, this piece of advice, true as it may be, isn’t much help for those of us who don’t have such systems. Perhaps someday a set of such locales may be standardized, but right now they aren’t. If someone hasn’t already written a case-insensitive comparison function for you, you’ll have to write it yourself.

Case-insensitive string comparison

Using `ctype`, it’s straightforward to build case-insensitive string comparison out of case-insensitive character comparison. This version isn’t optimal, but at least it’s correct. It uses essentially the same technique as before: compare two strings using `lexicographical_compare`, and compare two characters by converting both of them to uppercase. This time, though, we’re being careful to use a locale object instead of a global variable. (As an aside, converting both characters to uppercase might not always give the same results as converting both characters to lowercase: There’s no guarantee that the operations are inverses. In French, for example, it’s customary to omit accent marks on uppercase characters. In a French locale it might be reasonable for `toupper` to be a lossy conversion; it might convert both `'é'` and `'e'` to the same uppercase character, `'E'`. In such a locale, then, a case-insensitive comparison using `toupper` will say that `'é'` and `'E'` are equivalent characters while a case-insensitive comparison using `tolower` will say that they aren't. Which is the right answer? Probably the former, but it depends on the language, on local customs, and on your application.)

Click here to view code image

```
struct lt_str_1
  : public std::binary_function<std::string, std::string,
bool> {

  struct lt_char {
    const std::ctype<char>& ct;

    lt_char(const std::ctype<char>& c) : ct(c) {}

    bool operator()(char x, char y) const {
      return ct.toupper(x) < ct.toupper(y);
    }

  };

  std::locale loc;
  const std::ctype<char>& ct;


  lt_str_1(const std::locale& L = std::locale::classic())
    : loc(L), ct(std::use_facet<std::ctype<char> >(loc)) {}

  bool operator()(const std::string& x, const std::string& y)
const {
    return std::lexicographical_compare(x.begin(), x.end(),
                                        y.begin(), y.end(),
                                        lt_char(ct));
  }

};
```

This isn't quite optimal yet; it's slower than it ought to be. The problem is annoying and technical: we're calling `toupper` in a loop, and the C++ standard requires `toupper` to make a virtual function call. Some optimizers may be smart enough to move the virtual function overhead out of the loop, but most aren't. Virtual function calls in loops should be avoided.

In this case, avoiding it isn't completely straightforward. It's tempting to think that the right answer is another one of `ctype`'s member functions,

```
const char* ctype<char>::toupper(char* f, char* l) const,
```

which changes the case of the characters in the range `[f, l)`. Unfortunately, this isn't quite the right interface for our purpose. Using it to compare two strings would require copying both strings to buffers and then converting the buffers to uppercase. Where do those buffers come from? They can't be fixed size arrays (how large is large enough?), but dynamic arrays would require an expensive memory allocation.

An alternative solution is to do the case conversion once for every character, and cache the result. This isn't a fully general solution—it would be completely unworkable, for example, if you were working with 32-bit UCS-4 characters. If you're working with `char`, though (8 bits on most systems), it's not so unreasonable to maintain 256 bytes of case conversion information in the comparison function object.

Click here to view code image

```
struct lt_str_2 :
  public std::binary_function<std::string, std::string, bool>
{

  struct lt_char {
    const char* tab;

    lt_char(const char* t) : tab(t) {}

    bool operator()(char x, char y) const {
      return tab[x-CHAR_MIN] < tab[y - CHAR_MIN];
    }

  };

  char tab[CHAR_MAX - CHAR_MIN + 1];


  lt_str_2(const std::locale& L = std::locale::classic()) {
    const std::ctype<char>& ct =
std::use_facet<std::ctype<char> >(L);
```

```
    for (int i = CHAR_MIN; i <= CHAR_MAX; ++i)
      tab[i - CHAR_MIN] = (char) i;

    ct.toupper(tab, tab + (CHAR_MAX - CHAR_MIN + 1));
  }

  bool operator()(const std::string& x, const std::string& y)
const {
    return std::lexicographical_compare(x.begin(), x.end(),
                                        y.begin(), y.end(),
                                        lt_char(tab));
}

};
```

As you can see, `lt_str_1` and `lt_str_2` aren't very different. The former has a character-comparison function object that uses a `ctype` facet directly, and the latter a character-comparison function object that uses a table of precomputed uppercase conversions. This might be slower if you're creating an `lt_str_2` function object, using it to compare a few short strings, and then throwing it away. For any substantial use, though, `lt_str_2` will be noticeably faster than `lt_str_1`. On my system the difference was more than a factor of two: it took 0.86 seconds to sort a list of 23,791 words with `lt_str_1`, and 0.4 with `lt_str_2`.

What have we learned from all of this?

- A case-insensitive string class is the wrong level of abstraction. Generic algorithms in the C++ standard library are parameterized by policy, and you should exploit that fact.
- Lexicographical string comparisons are built out of character comparisons. Once you've got a case-insensitive character comparison function object, the problem is solved. (And you can reuse that function object for comparisons of other kinds of character sequences, like `vector<char>`, or string tables, or ordinary C strings.)
- Case-insensitive character comparison is harder than it looks. It's meaningless except in the context of a particular locale, so a character comparison function object needs to store locale information. If speed is a concern, you should write the function object to avoid repeated calls of expensive facet operations.

Correct case-insensitive comparison takes a fair amount of machinery, but you only have to write it once. You probably don’t want to think about locales; most people don’t. (Who wanted to think about Y2K bugs in 1990?) You stand a better chance of being able to ignore locales if you get locale-dependent code right, though, than if you write code that glosses over the dependencies.

Appendix B. Remarks on Microsoft's STL Platforms

In the opening pages of this book, I introduced the term *STL platform* to refer to the combination of a particular compiler and a particular implementation of the Standard Template Library. If you are using version 6 or earlier of a Microsoft Visual C++ compiler (i.e., a compiler that ships with version 6 or earlier of Microsoft's Visual Studio), the distinction between a compiler and a library is particularly important, because the compiler is sometimes more capable than the accompanying STL implementation suggests. In this appendix, I describe an important shortcoming of older Microsoft STL platforms and I offer workarounds that can significantly improve your STL experience.

The information that follows is for developers using Microsoft Visual C++ (MSVC) versions 4–6. If you're using Visual C++ .NET, your STL platform doesn't have the problems described below, and you may ignore this appendix.

Member Function Templates in the STL

Suppose you have two `vector`s of `Widget`s and you'd like to copy the `Widget`s in one `vector` to the end of `another`. That's easy. Just use `vector`'s range `insert` function (see Item 5):

Click here to view code image

```
vector<Widget> vw1, vw2;

...

vw1.insert(vw1.end(), vw2.begin(), vw2.end());    // append to
vw1 a copy
                                                  // of the
Widgets in vw2
```

If you have a `vector` and a `deque`, you do the same thing:

Click here to view code image

```
vector<Widget> vw;
```

```
deque<Widget> dw;

...

vw.insert(vw.end(), dw.begin(), dw.end());        // append to
vw a copy
                                                  // of the
Widgets in dw
```

In fact, you can do this no matter what the type of the container holding the objects to be copied. Even custom containers work:

Click here to view code image

```
vector<Widget> vw;
...

list<Widget> lw;
...

vw.insert(vw.begin(), lw.begin(), lw.end());    // prepend to
vw a copy
                                                // of the
Widgets in lw

set<Widget> sw;
...

vw.insert(vw.begin(), sw.begin(), sw.end());    // prepend to
vw a copy
                                                // of the
Widgets in sw

template<typename T,                            // template
for a custom
         typename Allocator = allocator<T> >    // STL-
compatible
class SpecialContainer { ... };                  // container

SpecialContainer<Widget> scw;
...

vw.insert(vw.end(), scw.begin(), scw.end());    // append to
vw a copy
                                                // of the
Widgets in scw
```

This flexibility is possible because `vector`’s range `insert` function isn’t a function at all. Rather, it’s a *member function template* that can be instantiated with *any iterator type* to generate a specific range `insert` function. For `vector`, the Standard declares the `insert` template like this:

Click here to view code image

```
template <class T, class Allocator = allocator<T> >
class vector {
public:

  ...

  template <class InputIterator>
  void insert(iterator position, InputIterator first,
InputIterator last);

  ...

};
```

Each standard container is required to offer this templatized range `insert`. Similar member function templates are required for the containers’ range constructors and for the range form of `assign` (both of which are discussed in Item 5).

MSVC Versions 4–6

Unfortunately, the STL implementation that ships with MSVC versions 4–6 declares no member function templates. This library was originally developed for MSVC version 4, and that compiler, like most compilers of its day, lacked member function template capabilities. Between MSVC4 and MSVC6, the compiler added support for these templates, but, due to legal proceedings that affected Microsoft without directly involving them, the library remained essentially frozen.

Because the STL implementation shipping with MSVC4–6 was designed for a compiler lacking member function templates, the library’s authors approximated the functionality of such templates by replacing each template with a specific function, one that accepted only iterators from the same container type. For `insert`, for example, the member function template was replaced with this:

Click here to view code image

```
void insert(iterator position,                   //
"iterator" is the
            iterator first, iterator last);      //
container's iterator type
```

This restricted form of range member functions makes it possible to perform a range `insert` from a `vector<Widget>` to a `vector<Widget>` or from a `list<int>` to a `list<int>`, but not from a `vector<Widget>` to a `list<Widget>` or from a `set<int>` to a `deque<int>`. It's not even possible to do a range `insert` (or `assign` or construction) from a `vector<long>` to a `vector<int>`, because `vector<long>::iterator` is not the same type as a `vector<int>::iterator`. As a result, the following perfectly valid code fails to compile using MSVC4–6:

Click here to view code image

```
istream_iterator<Widget> begin(cin), end;     // create begin
and end
                                              // iterators
for reading
                                              // Widgets from
cin
                                              // (see Item 6)

vector<Widget> vw(begin, end);                // read cin's
Widgets into vw
                                              // (again, see
Item 6); won't
                                              // compile with
MSVC4-6

list<Widget> lw;
...
lw.assign(vw.rbegin(), vw.rend());            // assign vw's
contents to lw
                                              // (in reverse
order); won't
                                              // compile with
MSVC4-6

SpecialContainer<Widget> scw;
...
scw.insert(scw.end(), lw.begin(), lw.end());  // insert at
the end of scw a
                                              // copy of the
```

```
Widgets in lw;
                                                  // won't
compile with
MSVC4-6
```

So what do you do if you must use MSVC4–6? That depends on the MSVC version you are using and whether you are forced to use the STL implementation that comes with the compiler.

A Workaround for MSVC4–5

Look again at the valid code examples that fail to compile with the STL that accompanies MSVC4–6:

Click here to view code image

```
vector<Widget> vw(begin, end);                       // rejected
by the MSVC4-6
                                                     //STL
implementation

list<Widget> lw;
...
lw.assign(vw.rbegin(), vw.rend());                   //also
rejected

SpecialContainer<Widget> scw;
...
scw.insert(scw.end(), lw.begin(), lw.end());         // ditto
```

These calls look rather different, but they all fail for the same reason: missing member function templates in the STL implementation. There’s a single cure for all of them: use `copy` and an insert iterator (see Item 30) instead. Here, for example, are the workarounds for the examples above:

Click here to view code image

```
istream_iterator<Widget> begin(cin), end;

vector<Widget> vw;                                     //
default-construct vw;
copy(begin, end, back_inserter(vw));                   // then
copy the
                                                       // Widgets
in cin into it
```

```
list<Widget> lw;
...
lw.clear();                                         //
eliminate lw's old
copy(vw.rbegin(), vw.rend(), back_inserter(lw));    //
Widgets; copy over
                                                    // vw's
Widgets (in
                                                    // reverse
order)

SpecialContainer<Widget> scw;
...
copy(lw.begin(), lw.end(),                          // copy
lw's Widgets to
     inserter(scw, scw.end()));                     // the end
of scw
```

I encourage you to use these `copy`-based workarounds with the library that ships with MSVC4–5, but beware! Don't fall into the trap of becoming so comfortable with the workarounds, you forget that they are *workarounds*. As Item 5 explains, using the `copy` algorithm is almost always inferior to using a range member function, so as soon as you have a chance to upgrade your STL platform to one that supports member function templates, stop using `copy` in places where range member functions are the proper approach.

An Additional Workaround for MSVC6

You can use the MSVC4–5 workaround with MSVC6, too, but for MSVC6 there is another option. The compilers that are part of MSVC4–5 offer no meaningful support for member function templates, so the fact that the STL implementation lacks them is immaterial. The situation is different with MSVC6, because MSVC6's compiler does support member function templates. It thus becomes reasonable to consider replacing the STL that ships with MSVC6 with an implementation that provides the member function templates the Standard prescribes.

Item 50 explains that both SGI and STLport offer freely downloadable STL implementations, and both of those implementations count the MSVC6 compiler as one with which they'll work. You can also purchase the latest MSVC-compatible STL implementation from Dinkumware. Each option has advantages and disadvantages.

SGI's and STLport's implementations are free, and I suspect you know what that means as regards official support for the software: there isn't any. Furthermore, because SGI and STLport design their libraries to work with a variety of compilers, you will probably have to manually configure their implementations to get the most out of MSVC6. In particular, you may have to explicitly enable support for member function templates, because, working with many compilers as they do, SGI and/or STLport may not enable that by default. You may also have to worry about linking with other MSVC6 libraries (especially DLLs), including things like making sure you use the appropriate builds for threading and debugging, etc.

If that kind of thing scares you, or if you've been known to grumble that you can't afford free software, you may want to look into Dinkumware's replacement library for MSVC6. It's designed to be drop-in compatible with the native MSVC6 STL and to maximize MSVC6's adherence to the Standard as an STL platform. Since Dinkumware authored the STL that ships with MSVC6, there's a decent chance their latest STL implementation really *is* a drop-in replacement. To learn more about Dinkumware STL implementations, visit the company's web site: http://www.dinkumware.com/.

Regardless of whether you choose SGI's, STLport's, or Dinkumware's implementation as an STL replacement, you'll do more than gain an STL with member function templates. You'll also bypass conformance problems in other areas of the library, such as `string` failing to declare `push_back`. Furthermore, you'll gain access to useful STL extensions, including hashed containers (see Item 25) and singly linked lists (`slist`s). SGI's and STLport's implementations also offer a variety of nonstandard functor classes, such as `select1st` and `select2nd` (see Item 50).

Even if you're trapped with the STL implementation that ships with MSVC6, it's probably worth your while to visit the Dinkumware web site. That site lists known bugs in the MSVC6 library implementation and explains how to modify your copy of the library to reduce its shortcomings. Needless to say, editing your library headers is something you do at your own risk. If you run into trouble, don't blame me.

Footnotes

[†] If you're interested in the tortured history of `auto_ptr` standardization, point your web browser to the *`auto_ptr` Update page* [29] at the *More Effective C++* web site.

[†] This is true only for the forms of `erase` that take iterator arguments. Associative containers also offer a form of `erase` taking a value argument, and that form returns the number of elements erased. Here, however, we're concerned only with `erase`ing things via iterators.

[†] There's a loophole in the guarantee. If an exception is not caught at all, the program will terminate. In that case, local objects (such as `lock`) may not have their destructors called. Some compilers call them, some do not. Both behaviors are valid.

[†] Because $\log_{1.5}1000 = 18$ (rounding up to an integer).

[†] See Andrei Alexandrescu's column in the April 2000 *C++ Report* [19].

Index

The example classes and class templates declared or defined in this book are indexed under *example classes/templates*. The example functions and function templates are indexed under *example functions/templates*.

Before A

`__default_alloc_template` 211C

A

abstraction bonus, 203C
abstraction penalty, 201C
`accumulate`
    function objects for, 158C
    initial value and, 157C, 159C
    side effects and, 160C
adaptability
    algorithm function objects and, 156C
    definition of, 170C
    functor classes and, 169–173C
    overloading `operator()` and, 173C
add or update functionality, in `map`, 107C
`adjacent_difference`, 157C
`advance`
    efficiency of, 122C
    to create `iterator`s from `const_iterator`s, 120–123C
Alexandrescu, Andrei, 227C, 230C
algorithms
    `accumulate` 156–161C
        function objects for, 158C
        initial value and, 157C, 159C
        side effects and, 160C

  - adaptable function objects and, 156C
  - `adjacent_difference`, 157C
  - as a vocabulary, 186C
  - `binary_search`, 192–201C

container mem funcs vs. 192C
`copy`, eliminating calls to, 26C
`copy_if`, implementing, 154–156C
copying func objects within, 166–168C
`count`, 156C, 192–201C
  - `equal_range` vs., for sorted ranges, 197C

`count_if`, 157C
efficiency, vs. explicit loops, 182–184C
`equal_range`, 192–201C
  - `count` vs., for sorted ranges, 197C

`find`, 192–201C
  - `count` in `multiset`, `multimap` vs. 199C
  - `lower_bound` in `multiset` , `multimap` vs. 201C
  - using equivalence vs. equality, 86C

`for_each`, 156–161C
  - side effects and, 160C

function call syntax in, 175C
function parameters to, 201–205C
hand-written loops vs. 181C–189C
`includes`, 148C
`inner_product`, 157C
`inplace_merge`, 148C
`lexicographical_compare`, 153C
longest name, 153C
loops and, 182C
`lower_bound`, 192–201C
  - equality vs. equivalence and, 195C

`max_element`, 157C
`merge`, 148C, 192C

`min_element`, 157C
`mismatch`, 151C
`nth_element`, 133–138C
optimizations in, 183C
`partial_sort`, 133–138C
`partial_sum`, 157C
`partition`, 133–138C
    containers of pointers and, 145C
`remove`, 139–143C
    on containers of pointers, 143–146C
`remove_copy_if`, 44C
`remove_if`, 44C, 144C, 145C
    see also `remove`
    possible implementation of, 167C
`set_difference`, 148C
`set_intersection`, 148C
`set_symmetric_difference`, 148C, 153C
`set_union`, 148C
`sort`, 133–138C
sorted ranges and, 146–150C
sorting, 133–138C
    alternatives to, 138C
`stable_partition`, 133–138C
`stable_sort`, 133–138C
`string` member functions vs. 230C
`transform`, 129C
`unique`, 145C, 148C, 149C
    see also `remove`
`unique_copy`, 148C, 149C
`upper_bound`, 197–198C
All your base are belong to us
allocations
    minimizing via `reserve`, 66–68C

- minimum, in `string`, 72C

`allocator`

- `allocate` interface, vs. `operator new`, 51C

allocators

- boilerplate code for, 54C
- conventions and restrictions, 48–54C
- fraud in interface, 51C
- in `string`, 69C
- legitimate uses, 54–58C
- never being called, 52C
- permitted assumptions about, 49C
- `rebind` template within, 53C
- stateful, portability and, 50C, 51C, 212C
- summary of guidelines for, 54C
- typedefs within, 48C
- URLs for examples, 227C, 228C

allusions

- to *Candide*, 143C
- to *Do Not Go Gentle into that Good Night*, 187C
- to Martin Luther King, Jr. 51C
- to Matthew, 25 222C
- to *The Little Engine that Could*, 63C

amortized constant time complexity, 6C

ANSI Standard for C++

- see C++ Standard, The

anteater, 88C

`argument_type`, 170C

array-based containers, definition of, 13C

arrays

- as part of the STL, 2C
- as STL containers, 64C
- containers vs. 22C
- `vector` and `string` vs. 63C–66C

assignments
    `assign` vs. `operator=`, 25C
    superfluous, avoiding, 31C
    via range member functions, 33C
associative containers
    see standard associative containers, hashed containers
Austern, Matt, 54C, 226C, 227C, 228C
    see also *Generic Programming and the STL*
`auto_ptr`
    as container element, 40–43C
    semantics of copying, 41C
    sorting, 41C
    URL for update page on, 228C
average, finding, for a range, 159–161C

B

`back_insert_iterator`, 216C
`back_inserter`, 130C, 216C
    `push_back` and, 130C
backwards order, inserting elements in, 184C
`base`, see `reverse_iterator`
`basic_ostream`, relation to `ostream`, 230C
`basic_string`
    relation to `string`, 210C, 229C
    relation to `string` and `wstring`, 4C
`begin`/`end`, relation to `rbegin`/`rend`, 123C
bidirectional iterators
    binary search algorithms and, 148C
    definition of, 5C
    standard associative containers and, 5C
`binary_function`, 170–172C
    pointer parameters and, 172C
    reference parameters and, 171C

`binary_search`, 199C
  - `bsearch` vs. 194C
  - related functions vs. 192–201C

`bind1st`, 216C
  - adaptability and, 170C

`bind2nd`, 210C, 216C
  - adaptability and, 170C
  - reference-to-reference problem and, 222C

`binder1st`, 216C

`binder2nd`, 216C

binders, definition of, 5C

bitfields, 80C

`bitset`
  - as alternative to `vector<bool>`, 81C
  - as part of the STL, 2C

Boost, 170C, 172C, 221–223C
  - `shared_array`, 222C
  - `shared_ptr`, 39C, 146C, 165C, 178C, 222C
  - web site URL, 217C, 227C

Bridge Pattern, 165C

`bsearch`, vs. `binary_search`, 194C

bugs
  - Dinkumware list for MSVC, 244C

Bulka, Dov, 226C
  - see also *Efficient C++*

C

C++ Programming Language, The, 61C, 155C
  - bibliography entry for, 226C

C++ Standard Library, The, 2C, 6C, 94C, 113C, 207C
  - bibliographic entry for, 225C

C++ Standard, The
  - bibliography entry for, 226C

- citation number omitted for, 3C
- guidance on choosing containers, 12C
- reference counting and, 64C
- URL for purchasing, 226C

Candide , allusion to, 143C

capacity
- cost of increasing for `vector`/`string`, 66C
- minimizing in `vector` and `string`, 77–79C

`capacity`, vs. `size`, 66–67C

case conversions, when not one-for-one, 234C

case-insensitive
- string class, 229–230C
- `string` comparisons, see `string`

casts
- `const_iterator` to `iterator`, 120C
- creating temporary objects via, 98C
- to references, 98C
- to remove constness, 98C
- to remove `map`/`multimap` constness, 99C

categories, for iterators, 5C

`char_traits`, 113C, 211C, 230C

choosing
- among `binary_search`, `lower_bound` , `upper_bound` , and `equal_range`, 192–201C
- among containers, 12C
- among iterator types, 116–119C
- `vector` vs. `string`, 64C

citations, in this book, 2C

`class` vs. `typename`, 7C

classes, vs. structs for functor classes, 171C

clustering, in node-based containers, 103C

COAPs, see containers, `auto_ptr` `collate` facet, 234C

color, use in this book, 8C

comparison functions
    consistency and, 149C
    equal values and, 92–94C
    for `pair`s, 104C
comparisons
    see also string, comparisons, comparison functions
    `iterator`s with `const_iterator`s, 118C
    lexicographic, 231C
compilers
    diagnostics, deciphering, 210–217C
    implementations, vs. STL impls, 3C
    independence, value of, 3C
    optimizations, inlining and, 202C
    problems, see workarounds
complexity
    amortized constant time, 6C
    constant time, 6C
    guarantees, in the STL, 5–6C
    linear time, 6C
    logarithmic time, 6C
`compose1`, 219C
`compose2`, 187C, 219C
`const_iterator`
    casting to `iterator`, 120C
    comparisons with `iterator`s, 118C
    converting to `iterator`, 120–123C
    other iterator types vs. 116–119C
`const_reverse_iterator`
    other iterator types vs. 116–119C
constant time complexity, 6C
    amortized, see amortized constant time
constness
    casting away, 98C

- of `map`/`multimap` elements, 95C
- of `set`/`multiset` elements, 95C

construction, via range mem funcs, 31C

container adapters, as part of the STL, 2C

container-independent code, 16C, 47C
- illusory nature of, 15–20C

containers
- arrays as, 64C
- arrays vs. 22C
- `assign` vs. `operator=` in, 25C
- associative, see standard associative containers
- `auto_ptr`s in, 40–43C
- calling `empty` vs. `size`, 23–24C

choosing among
- advice from the Standard, 12C
- iterator types, 116–119C

contiguous memory
- see contiguous-memory containers,`vector`, `string`, `deque`, `rope`

converting `const_iterator` s to `iterator`s, 120–123C

criteria for selecting, 11–15C

deleting pointers in, 36–40C

encapsulating, 19C

erasing
- see also `erase`-`remove` idiom
- elements in, 43–48C
- iterator invalidation during, 14C, 45C
- relation to `remove`, 139–143C

exception safety and, 14C, 37C, 39C

filling from legacy APIs, 77C

hashed, see hashed containers

improving efficiency via `reserve`, 66–68C

insertions
- in reverse order, 184C

  - iterator invalidation during, 14C
- iterators
  - casting among, 121C
  - invalidation, see iterators, invalidation
- mem funcs vs. algorithms, 190–192C
- node-based
  - see node-based containers, `list` , standard associative containers, `slist`
- object copying and, 20–22C
- of pointers, `remove` and, 143–146C
- of proxy objects, 82C
- of smart pointers, 39C
- range vs. single-element member functions, 24–33C
- relation of `begin`/`end`, `rbegin`/`rend`, 123C
- replacing one with another, 18C
- requirements in the Standard, 79C
- `resize` vs. `reserve`, 67C
- rolling back insertions and erasures, 14C
- `rope`, 218C
- sequence
  - see standard sequence containers
- size vs. capacity, 66–67C
- `size_type` typedef, 158C
- `slist`, 218C
- sorted `vector` vs. associative, 100–106C
- thread safety and, 58–62C
- transactional semantics for insertion and erasing, 14C
- typedefs for, 18–19C
- `value_type` typedef, 36C, 108C
- `vector<bool>`, problems with, 79–82C
- contiguous memory
  - for `string`, 75C
  - for `vector`, 74C
- contiguous-memory containers

  - see also `vector`, `string`, `deque`, `rope` cost of erasing in, 32C
  - cost of insertion, 28C
  - definition of, 13C
  - iterator invalidation in, see iterators, invalidation
- conventions, for allocators, 48–54C
- conversions
  - among iterator types, 117C
  - from `const_iterator` to `iterator`, 120–123C
  - when not one-for-one, 234C
- `copy`
  - eliminating calls to, 26C
  - insert iterators and, 26C
  - missing member templates and, 242C
- `copy_if`, implementing, 154–156C
- copying
  - `auto_ptr`s, semantics of, 41C
  - function objects
    - efficiency and, 164C
    - within algorithms, 166–168C
  - objects in containers, 20–22C
- `count`, 156C
  - as existence test, 193C
  - `equal_range` vs., for sorted ranges, 197C
  - related functions vs. 192C–201C
- `count_if`, 157C
- `<cctype>`, conventions of functions in, 151C
- `<ctype.h>` , conventions of functions in, 151C

D

- debug mode, see STLport, debug mode
- deciphering compiler diagnostics, 210–217C
- definitions
  - adaptable function object, 170C

amortized constant time complexity, 6C
array-based container, 13C
bidirectional iterator, 5C
binder, 5C
COAP, 40C
constant time complexity, 6C
container-independent code, 16C
contiguous-memory container, 13C
equality, 84C
equivalence, 84–85C
forward iterator, 5C
function object, 5C
functor, 5C
functor class, 5C
input iterator, 5C
linear time complexity, 6C
local class, 189C
logarithmic time complexity, 6C
monomorphic function object, 164C
node-based container, 13C
output iterator, 5C
predicate, 166C
predicate class, 166C
pure function, 166C
random access iterator, 5C
range member function, 25C
resource acquisition is initialization, 61C
stability, in sorting, 135C
standard associative container, 5C
standard sequence container, 5C
STL platform, 3C
transactional semantics, 14C

`delete`

  - `delete[]` vs. 63C
  - using wrong form, 64C
- deleting
  - objects more than once, 64C
  - pointers in containers, 36–40C
- dependent types, `typename` and, 7–8C
- `deque`
  - unique invalidation rules for, 15C
    - see also iterators, invalidation
- `deque<bool>` , as alternative to `vector<bool>`, 81C
- dereferencing functor class, 90C
- *Design Patterns*, 80C, 165C
  - bibliography entry for, 226C
- *Design Patterns CD*
  - bibliography entry for, 226C
- destination range, ensuring adequate size, 129–133C
- destructors, calling explicitly, 56C
- dictionary order comparison, see lexicographic comparison
- Dinkumware
  - bug list for MSVC STL, 244C
  - hashed containers implementation, 114C
  - interface for hashed containers, 113C
  - `slist` implementation, 218C
  - STL replacement for MSVC6 243C
  - web site for, 243C
- disambiguating function and object declarations, 35C
- `distance`
  - declaration for, 122C
  - efficiency of, 122C
  - explicit template argument specification and, 122C
  - to convert `const_iterator` s to `iterator`s, 120–123C
- *Do Not Go Gentle into that Good Night* , allusion to, 187C
- documentation, on-line, for the STL, 217C

E

Effective C++
  bibliography entry for, 225C
  citation number omitted for, 3C
  inheritance from class without a virtual destructor discussion in, 37C
  inlining discussion in, 202C
  object copying discussions in, 21C
  pointer to implementation class discussion in, 165C
  slicing problem discussion in, 22C
  URL for errata list for, 228C
Effective C++ CD
  bibliography entry for, 225C
  URL for errata list for, 228C
efficiency
  see also optimizations
  `advance` and, 122C
  algorithms vs. explicit loops, 182–184C
  associative container vs. sorted `vector`, 100–106C
  case-insensitive string comparisons and, 154C
  comparative, of sorting algorithms, 138C
  `copy` vs. range `insert`, 26C
  copying
    function objects and, 164C
    objects and, 21C
`distance` and, 122C
`empty` vs. `size`, 23–24C
  erasing from contiguous-memory containers, 32C
  function objects vs. functions, 201–205C
  hashed containers, overview of, 111–115C
  hashing and, 101C
  “hint” form of `insert` and, 110C
  improving via custom allocators, 54C

  - increasing `vector`/`string` capacity, 66C
  - inlining and, 202C
  - inserting into contiguous-memory containers, 28C
  - `istreambuf_iterator`s and, 126–127C
  - Items on, 9C
  - `list::remove` vs. the `erase`-`remove` idiom, 43C
  - logarithmic vs. linear, 190C
  - `map::operator[]` vs. `map::insert`, 106–111C
  - mem funcs vs. algorithms, 190–192C
  - minimizing reallocs via `reserve`, 66–68C
  - `ostreambuf_iterator`s and, 127C
  - range vs. single-element member functions, 24–33C
  - small string optimization, 71C
  - `sort` vs. `qsort`, 203C
  - sorting algorithms, overview of, 133–138C
  - `string` implementation trade-offs, 68–73C
  - `toupper` and, 236–237C
  - `use_facet` and, 234C
- *Efficient C++*, 202C
  - bibliography entry for, 226C
- Einstein, Albert, 69C
- Emacs, 27C
- email address
  - for the President of the USA, 212C
- embedded nulls, 75C, 154C
- `empty`, vs. `size`, 23–24C
- encapsulating containers, 19C
- `equal_range`, 196–197C
  - `count` vs., for sorted ranges, 197C
  - related functions vs. 192–201C
- `equal_to`, 86C, 112C
- equality
  - definition of, 84C

- equivalence vs. 83–88C
  - in hashed containers, 113C
- `lower_bound` and, 195C

equivalence
- definition of, 84–85C
- equality vs. 83–88C
  - in hashed containers, 113C

`lower_bound` and, 195C

equivalent values, inserting in order, 198C

`erase`

see also `erase`-`remove` idiom
- relation to `remove` algorithm, 139–143C
- return types for, 32C
- return value for standard sequence containers, 46C

`erase_after`, 218C

`erase`-`remove` idiom, 43C, 47C, 142C, 145C, 146C, 184C, 207C
- limitations of, 46C

`list::remove` vs. 43C

`remove_if` variant, 144C

erasing
- see also containers, erasing
- `base` iterators and, 124C
- elements in containers, 43–48C
- rolling back, 14C
- via range member functions, 32C

errata list
- for *Effective C++*, 228C
- for *Effective C++ CD*, 228C
- for *More Effective C++*, 228C

error messages, deciphering, 210–217C

example classes/templates
- `Average`, 205C
- `BadPredicate`, 167C, 168C

`BetweenValues`, 188C, 189C
`BPFC`, 164C, 165C
`BPFCImpl`, 165C
`CIStringCompare`, 85C
`Contestant`, 77C
`CustomerList`, 20C
`DataCompare`, 105C
`DeleteObject`, 37C, 38C
`Dereference`, 90C
`DereferenceLess`, 91C
`DoSomething`, 163C
`Employee`, 95C
`Heap1`, 57C
`Heap2`, 57C
`IDNumberLess`, 96C
`list`, 52C
`list::ListNode`, 52C
`Lock`, 60C
`lt_nocase`, 231C
`lt_str_1`, 235C
`lt_str_1::lt_char`, 235C
`lt_str_2`, 236C
`lt_str_2::lt_char`, 236C
`MaxSpeedCompare`, 179C
`MeetsThreshold`, 171C
`NiftyEmailProgram`, 212C, 215C
`Person`, 198C
`PersonNameLess`, 198C
`Point`, 159C, 161C
`PointAverage`, 160C, 161C
`PtrWidgetNameCompare`, 172C
`RCSP`, 146C
`SharedMemoryAllocator`, 55C

`SpecialAllocator`, 19C, 49C
`SpecialContainer`, 240C
`SpecialString`, 37C
`SpecialWidget`, 21C
`SpecificHeapAllocator`, 57C
`std::less<Widget>`, 178C
`StringPtrGreater`, 93C
`StringPtrLess`, 89C
`StringSize`, 204C
`Timestamp`, 197C
`vector`, 240C
`Widget`, 7C, 18C, 19C, 21C, 35C, 84C, 106C, 111C, 143C, 174C, 177C, 182C, 222C
`WidgetNameCompare`, 171C

example functions/templates
`anotherBadPredicate`, 169C
`average`, 204C
`Average::operator()`, 205C
`BadPredicate::BadPredicate()`, 167C
`BadPredicate::operator()`, 167C, 168C
`BetweenValues::BetweenValues`, 188C
`BetweenValues::operator()`, 188C
`BPFC::operator()`, 164C, 165C
`BPFCImpl::“BPFCImpl`, 165C
`BPFCImpl::operator()`, 165C
`ciCharCompare`, 151C
`ciCharLess`, 153C
`ciStringCompare`, 152C, 153C, 154C
`CIStringCompare::operator()`, 85C
`ciStringCompareImpl`, 152C
`copy_if`, 155C, 156C
`DataCompare::keyLess`, 105C
`DataCompare::operator()`, 105C

delAndNullifyUncertified, 145C
DeleteObject::operator(), 37C, 38C
Dereference::operator(), 90C
DereferenceLess::operator(), 91C
doSomething, 36C, 37C, 38C, 39C, 74C, 75C, 77C
DoSomething::operator(), 163C
doubleGreater, 202C
efficientAddOrUpdate, 110C
Employee::idNumber, 95C
Employee::name, 95C
Employee::setName, 95C
Employee::setTitle, 95C
Employee::title, 95C
fillArray, 76C, 77C, 184C
fillString, 76C
hasAcceptableQuality, 137C
Heap1::alloc, 57C
Heap1::dealloc, 57C
IDNumberLess::operator(), 96C
isDefective, 155C
isInteresting, 169C
lastGreaterThanFirst, 8C
Lock::“Lock, 60C
Lock::Lock, 60C
lt_nocase::operator(), 231C
lt_str_1::lt_char::lt_char, 235C
lt_str_1::lt_char::operator(), 235C
lt_str_1::lt_str_1, 236C
lt_str_1::operator(), 236C
lt_str_2::lt_char::lt_char, 236C
lt_str_2::lt_char::operator(), 236C
lt_str_2::lt_str_2, 237C
lt_str_2::operator(), 237C

`MaxSpeedCompare::operator()`, 179C

`MeetsThreshold::MeetsThreshold`, 171C

`MeetsThreshold::operator()`, 171C

`NiftyEmailProgram::showEmailAddress`, 212C, 215C

`operator<` for `Timestamp`, 197C

`operator<` for `Widget`, 177C

`operator==` for `Widget`, 8C, 84C

`Person::name`, 198C

`Person::operator()`, 198C

`Point::Point`, 159C

`PointAverage::operator()`, 160C, 161C

`PointAverage::PointAverage`, 160C, 161C

`PointAverage::result`, 161C

`print`, 90C

`PtrWidgetNameCompare::operator()` 172C

`qualityCompare`, 134C

`SharedMemoryAllocator::allocate`, 55C

`SharedMemoryAllocator::deallocate`, 55C

`SpecificHeapAllocator::allocate`, 57C

`SpecificHeapAllocator::deallocate`, 57C

`std::less<Widget>::operator()`, 178C

`stringLengthSum`, 158C

`StringPtrGreater::operator()` 93C

`stringPtrLess`, 91C

`StringPtrLess::operator()`, 89C

`StringSize::operator()`, 204C

`test`, 174C

`transmogrify`, 129C, 220C

`vector<bool>::operator[]`, 80C

`vector<bool>::reference`, 80C

`Widget::isCertified`, 143C

`Widget::maxSpeed`, 177C

`Widget::operator=`, 21C, 106C, 111C

- `Widget::readStream`, 222C
- `Widget::redraw`, 182C
- `Widget::test`, 174C
- `Widget::weight`, 177C
- `Widget::Widget`, 21C, 106C
- `widgetAPCompare`, 41C
- `WidgetNameCompare::operator()`, 171C
- `writeAverages`, 204C, 205C

exception safety, 14C, 37C, 39C, 50C, 61C

Exceptional C++, 14C, 165C
- bibliography entry for, 226C

exceptions, to guidelines in this book, 10C

explicit template argument specification
- `distance` and, 122C
- `for_each` and, 163C
- `use_facet` and, 234C

extending the STL, 2C

F

facets, locales and, 234–235C

`find`
- `count` in `multiset`, `multimap` vs. 199C
- `lower_bound` in `multiset`, `multimap` vs. 201C
- related functions vs. 192–201C
- using equivalence vs. equality, 86C

`first_argument_type`, 170C

`for_each` declaration for, 163C
- explicit template argument specification and, 163C
- possible implementation of, 174C
- side effects and, 160C

forward iterators
- definition of, 5C
- `operator--` and, 5C

fragmentation, memory, reducing, 54C
fraud, in allocator interface, 51C
free STL implementations, 217C, 220C
`front_insert_iterator`, 216C
`front_inserter`, 130C, 216C
    `push_front` and, 130C
function objects
    as workaround for compiler problems, 204C
    definition of, 5C
    dereferencing, generic, 90C
    for `accumulate`, 158C
    functions vs. 201–205C
    monomorphic, definition of, 164C
    pass-by-value and, 162–166C
    slicing, 164C
functional programming, 206C
functions
    calling forms, 173C
    calling syntax in the STL, 175C
    comparison
        equal values and, 92–94C
        for pointers, 88–91C
    declaration forms, 33–35C
    declaring templates in, 188C
    function objects vs. 201–205C
    in `<cctype>`, conventions of, 151C
    in `<ctype.h>`, conventions of, 151C
    pointers to, as formal parameters, 34C
    predicates, need to be pure, 166–169C
    pure, definition of, 166C
    range vs. single-element, 24–33C
functor classes
    adaptability and, 169–173C

  - classes vs. structs, 171C
  - definition of, 5C
  - overloading `operator()` in, 114C
  - pass-by-value and, 162–166C
- functor, see function objects

G

- Gamma, Erich, 226C
  - see also *Design Patterns*
- *Generic Programming and the STL*, 2C, 94C, 217C, 229C
  - bibliography entry for, 226C
- gewürztraminer, 232C
- glass, broken, crawling on, 95C, 97C
- growth factor, for `vector`/`string`, 66C

H

- hand-written loops
  - algorithms vs. 181C–189C
  - iterator invalidation and, 185C
- hashed containers, 111–115C
  - Dinkumware interface for, 113C
  - equality vs. equivalence in, 113C
  - SGI interface for, 112C
  - two implementation approaches to, 114C
- headers
  - `#include`ing the proper ones, 209–210C
  - <algorithm> 210C
  - `<cctype>`, 151C
  - `<ctype.h>`, 151C
  - <functional> 210C
  - <iterator> 210C
  - <list> 209C

- <map> 209C
- <numeric> 210C
- `<numeric>`, 157C
- <set> 209C
- <vector> 209C
- failing to `#include`, 217C
- summary of, 209–210C

heaps, separate, allocators and, 56–58C

Helm, Richard, 226C

- see also *Design Patterns*

“hint” form of `insert`, 100C, 110C

I

identifiers, reserved, 213C

`identity`, 219C

implementations

- compilers vs. the STL, 3C
- variations for `string`, 68–73C

`includes`, 148C

`#include`s, portability and, 209–210C

inlining, 202C

- function pointers and, 203C

`inner_product`, 157C

`inplace_merge`, 148C

input iterators

- definition of, 5C
- range `insert` and, 29C

`insert`

- as member template, 240C
- “hint” form, 100C, 110C
- `operator[]` in `map` vs. 106–111C
- return types for, 32C

insert iterators

see also `inserter`, `back_inserter`, `front_inserter`
    container ::`reserve` and, 131C
    `copy` algorithm and, 26C
`insert_after`, 218C
`insert_iterator`, 216C
`inserter`, 130C, 216C
inserting
    see also containers, insertions
    `base` iterators and, 124C
    equivalent values in order, 198C
    in reverse order, 184C
    rolling back, 14C
    via range member functions, 32C
internationalization
    see also locales
    `strcmp` and, 150C
    `stricmp`/`strcmpi` and, 154C
invalidation, see iterators, invalidation
`ios::skipws`, 126C
iostreams library, SGI implementation of, 220C
ISO Standard for C++
    see C++ Standard, The
`istream_iterator`s, 157C, 210C
    `operator>>` and, 126C
    parsing ambiguities and, 35C
`istreambuf_iterator`s, 157C, 210C
    use for efficient I/O, 126–127C
Items on efficiency, list of, 9C
`iterator`
    comparisons with `const iterator`s, 118C
    other iterator types vs. 116–119C
    `reverse_iterator`'s base and, 123–125C
`iterator_traits`, 113C

  - `value_type` typedef, 42C
- iterators
  - see also `istreambuf_iterators`, `ostreambuf_iterators`
  - `base`, erasing and, 124C
  - `base`, insertion and, 124C
  - casting, 120C
  - categories, 5C
    - see also input iterators, output iterator, forward iterators, bidirectional iterators, random access iterators
    - in hashed containers, 114C
  - choosing among types, 116–119C
  - conversions among types, 117C
  - dereferencing function object for, 91C
  - implemented as pointers, 120C
  - invalidation
    - during `deque`::`insert`, 185C
    - during erasing, 14C, 45C, 46C
    - during insertion, 14C
    - during `vector` reallocation, 59C
    - during `vector`/`string insert`, 68C
    - during `vector`/`string` reallocation, 66C
    - during `vector`::`insert`, 27C
    - in hand-written loops, 185C
    - in standard sequence containers, 17C
    - in STLport STL implementation, 221C
    - predicting in `vector`/`string`, 68C
    - undefined behavior from, 27C, 45C, 46C, 185C
    - unique rules for `deque`, 15C
  - pointers in `vector` vs. 75C
  - relationship between `iterator` and `reverse_iterator`’s base, 123–125C
  - typedefs for, 18–19C
  - types in containers
    - see also `iterator` , `const_iterator` , `reverse_iterator`,

`const_reverse_iterator`
        casting among, 121C
    types, mixing 119C

J

Johnson, Ralph, 226C
see also *Design Patterns*
Josuttis, Nicolai, 225C, 227C
    see also *C++ Standard Library, The*

K

`key_comp`, 85C, 110C
keys, for `set`/`multiset`, modifying, 95–100C
King, Martin Luther, Jr., allusion to, 51C
Kreft, Klaus, 228C

L

Langer, Angelika, 228C
leaks, see resource leaks
legacy APIs
    filling containers from, 77C
    sorted `vector`s and, 76C
    `vector` and `string` and, 74–77C
    `vector<bool>` and, 81C
lemur, 88C
`less`, 210C
    `operator<` and, 177–180C
`less_equal`, 92C
lexicographic comparison, 231C
`lexicographical_compare`, 153C
    `strcmp` and, 153C
    use for case-insensitive string comparisons, 150–154C
`lhs`, as parameter name, 8–9C

linear time complexity, 6
    for binary search algorithms with bidirectional iterators, 148C
    logarithmic complexity vs. 190C
`list`
    algorithm specializations, 192C
    iterator invalidation in, see iterators, invalidation
    `merge`, 192C
    `remove`, 142–143C
        vs. the `erase`-`remove` idiom, 43C
    `sort`, 137C
    `splice`
        exception safety of, 50C
        vs. `size`, 23–24C
    `unique`, 143C
Little Engine that Could, The , allusion to, 63C
local classes
    definition of, 189C
    type parameters and, 189C
locales, 232–233C
    case-insensitive string comparisons and, 229–237C
    facets and, 234–235C
locality of reference, 103C
    improving via allocators, 55C
locking objects, 60C
logarithmic time complexity, 6C
    linear complexity vs. 190C
    meaning for binary search algorithms, 147C
longest algorithm name, 153C
lookup speed, maximizing, 100C
`lower_bound`
    equality vs. equivalence and, 195C
    related functions vs. 192–201C

M

`map`
- add or update functionality in, 107C
- constness of elements, 95C
- iterator invalidation in, see iterators, invalidation
- key, casting away constness, 99C
- `key_comp` member function, 110C
- `value_type` typedef, 108C

Matthew, 25:32, allusion to, 222C

`max_element`, 157C

`max_size`, 66C

Mayhew, David, 226C
- see also *Efficient C++*

`mem_fun`
- declaration for, 175C
- reasons for, 173–177C
- reference-to-reference problem and, 222C

`mem_fun_ref`
- reasons for, 173–177C
- reference-to-reference problem and, 222C

`mem_fun_ref_t`, 175C

`mem_fun_t`, 175C

member funcs, vs. algorithms, 190–192C

member function templates
- see member templates

member templates
- avoiding client redundancy with, 38C
- in the STL, 239–240C
- Microsoft's STL platforms and, 239–244C
- `vector::insert` as, 240C
- workaround for when missing, 242C

memory fragmentation, allocators and, 54C

memory layout
- for `string`, 69–71C, 75C
- for `vector`, 74C

memory leaks, see resource leaks
memory, shared, allocators and, 55–56C
`merge`, 148C, 192C
Microsoft's STL platforms, 239–244C
    Dinkumware replacement library for, 243C
`min_element`, 157C
`mismatch`, 151C
    use for case-insensitive string comparisons, 150–154C
mixing iterator types, 119C
modifying
    components in `std`, 178C
    `const` objects, 99C
    `set` or `multiset` keys, 95–100C
monomorphic function objects, 164C
More Effective C++
    `auto_ptr` and, 40C
    bibliography entry for, 225C
    citation number omitted for, 3C
    errata list, 228C
        smart pointers and, 39C
    placement `new` discussion in, 56C
    proxy objects discussion in, 49C, 80C
    reference counting discussion in, 4C, 71C
    resource acquisition is initialization discussion in, 61C
    smart pointer discussion in, 39C
    URL for `auto_ptr` update page for, 228C
    URL for errata list for, 228C
More Exceptional C++
    bibliography entry for, 226C
`multimap`
    constness of elements, 95C
    `find` vs. `count` in, 199C
    `find` vs. `lower_bound` in, 201C

- indeterminate traversal order in, 87C
- iterator invalidation in, see iterators, invalidation
- key, casting away constness, 99C
- `value_type` typedef, 108C

multiple deletes, 64C

`multiplies`, 159C

`multiset`
- constness of elements, 95C
- corrupting via element modification, 97C
- `find` vs. `count` in, 199C
- `find` vs. `lower_bound` in, 201C
- indeterminate traversal order in, 87C
- iterator invalidation in, see iterators, invalidation
- keys, modifying, 95–100C

multithreading
- allocators and, 54C
- containers and, 58–62C
- reference counting and, 64–65C
- `string` and, 64–65C

N

node-based containers
- see also standard associative containers, `list`, `slist` , hashed containers
- allocators and, 52C
- clustering in, 103C
- definition of, 13C

nonstandard containers
- see hashed containers, `slist`, `rope`

`not1`, 155C, 156C, 169C, 170C, 172C, 210C, 222C
- adaptability and, 170C

`not2`, 152C, 222C
- adaptability and, 170C

`nth_element`, 133–138C

nulls, embedded, 75C, 154C
`<numeric>`, 157C

O

objects
    copying, in containers, 20–22C
    for locking, 60C
    slicing, 21–22C, 164C
    temporary, created via casts, 98C
One True Editor, the, see Emacs
`operator new`, interface, vs.`allocator::allocate`, 51C
`operator()`, 5C
    declaring `const`, 168C
    functor class and, 5C
    inlining and, 202C
    overloading
        adaptability and, 173C
        in functor classes, 114C
`operator++`, side effects in, 45C
`operator--`, forward iterators and, 5C
`operator.` (“operator dot”) 49C
`operator<`, `less` and, 177–180C
`operator>>`
    `istream_iterator`s and, 126C
    `sentry` objects and, 126C
    whitespace and, 126C
`operator[]`, vs. `insert` in `map`, 106–111C
optimizations
    algorithms and, 183C
    function pointers and, 203C
    inlining and, 202C
    `istreambuf_iterator`s and, 127C
    range insertions and, 31C

  - reference counting and, 64C
  - small strings and, 71C
  - `stricmp`/`strcmpi` and, 154C
  - to reduce default allocator size, 70C
- `ostream`, relation to `basic_ostream`, 230C
- `ostream_iterator`s, 216C
- `ostreambuf_iterator`s, 216C
  - efficiency and, 127C
- output iterator, definition of, 5C
- overloading, `operator()` in functor classes, 114C

P

- page faults, 102C, 103C
- `pair`, comparison functions for, 104C
- parameters
  - function objects vs. functions, 201–205C
  - pointers to functions, 34C
  - type, local classes and, 189C
- parentheses
  - ignored, around parameter names, 33C
  - to distinguish function and object declarations, 35C
- parse, most vexing in C++ 33–35C
- parsing, objects vs. functions, 33–35C
- `partial_sort`, 133–138C
- `partial_sum`, 157C
- `partition`, 133–138C
  - containers of pointers and, 145C
  - `remove` vs. 141C
- pass-by-value
  - function objects and, 163C
  - functor classes and, 162–166C
- penguin, 88C
- Perfect Woman, see Woman, Perfect

performance, see efficiency
Pimpl Idiom, 165C
placement `new`, 56C
Plauger, P. J. 114C, 227C
pointers
  allocator typedef for, 48C
  as iterators, 120C
  as return type from `vector::begin`, 75C
  assignments, avoiding superfluous, 31C
  comparison functions for, 88–91C
  deleting in containers, 36–40C
  dereferencing function object for, 91C
  destructors for, 36C
  invalidation, see iterators, invalidation
  iterators in `vector` vs. 75C
  parameters, `binary_function` and, 172C
  parameters, `unary_function` and, 172C
  returned from `reverse_iterator::base`, 125C
  smart, see smart pointers
  to bitfields, 80C
  to functions, as formal parameters, 34C
portability
  `#include`s and, 209–210C
  casting `const` iterator to iterators, 121C
  container `::<auto_ptr> and`, 41C
  explicit template argument specification and, 163C
  hashed containers, code using, 112C
  `identity`, `project1st`, `project2nd`, `compose1`, `compose2`, `select1st`, `select2nd` and, 219C
  multiple compilers and, 3C
  range construction with `istream_iterator`s and, 35C
  `reverse_iterator::base` and, 125C
  `set`/`multiset` key modification and, 98C

  - stateful allocators and, 50C, 51C
  - STLport STL implementation and, 220C
  - `stricmp`/`strcmpi` and, 154C
- predicate class, definition of, 166C
- predicates
  - definition of, 166C
  - need to be pure functions, 166–169C
- predicting iterator invalidation, in `vector`/`string`, 68C
- principle of least astonishment, the, 179C
- `priority_queue`, 138C
  - as part of the STL, 2C
- `project1st`, 219C
- `project2nd`, 219C
- proxy objects, 49C
  - containers of, 82C
  - `vector<bool>` and, 80C
- `ptr_fun`, reasons for, 173–177C
- pure function, definition of, 166C
- `push_back`, `back_inserter` and, 130C
- `push_front`, `front_inserter` and, 130C

Q

- `qsort`, 162C
  - declaration for, 162C
  - `sort` vs. 203C
- `queue`, as part of the STL, 2C

R

- random access iterators
  - definition of, 5C
  - sorting algorithms requiring, 137C
- range

- destination, ensuring adequate size, 129–133C
- member functions, 25C
  - input iterators and, 29C
  - single-element versions vs. 24–33C
  - summary of, 31–33C
  - pointer assignments in `list` and, 31C
- sorted, algorithms requiring, 146–150C
- summarizing, 156–161C

`raw_storage_iterator`, 52C
RB trees, see red-black trees
`rbegin`/`rend`, relation to `begin`/`end`, 123C
reallocations
- invalidation of iterators during, 59C
- minimizing via `reserve`, 66–68C

rebinding allocators, 53C
red-black trees, 190C, 191C, 214C
redundant computations, avoiding via algorithm calls, 182C
Reeves, Jack, 227C
reference counting
- disabling, for `string`, 65C
- multithreading and, 64–65C
- smart pointers, 39C, 146C
  - see also Boost, `shared_ptr`
- `string` and, 64–65C
- The C++ Standard and, 64C
- this book and, 4C

references
- allocator typedef for, 48C
- casting to, 98C
- invalidation, see iterators, invalidation
- parameters, `binary_function` and, 171C
- parameters, `unary_function` and, 171C
- to bitfields, 80C

reference-to-reference problem, 222C
`remove`, 139–142C
  see also `erase-remove` idiom
  on containers of pointers, 143–146C
  `partition` vs. 141C
`remove_copy_if`, 44C
`remove_if`, 44C, 144C, 145C
  see also `remove`
  possible implementation of, 167C
`replace_if`, 186C
replacing STL implementations, 243C
`reserve`
  insert iterators and, 131C
  `resize` vs. 67C
`resize`
  reallocation and, 67C
  `reserve` vs. 67C
resource acquisition is initialization, 61C
resource leaks, 36C, 39C, 63C, 144C, 145C
  avoiding via smart pointers, 39C, 146C
  preventing via classes, 61C
`result_type`, 170C
return type
  `allocator::allocate` vs. `operator new`, 51C
  for *container* ::`begin`, 75C
  for `distance`, 122C
  for `erase`, 32C, 117C
  for function objects for `accumulate`, 158C
  for `insert`, 17C, 32C, 117C
  for `vector::operator[]`, 80C
reverse order
  inserting elements in, 184C
`reverse_iterator`
  `base` member function, 123–125C

- other iterator types vs. 116–119C

`rhs`, as parameter name, 8–9C

rolling back, insertions and erasures, 14C

`rope`, 218C

S

Scheme, 206C

`second_argument_type`, 170C

`select1st`, 219C

`select2nd`, 219C

`sentry` objects, `operator<<` and, 126C

separate heaps, allocators and, 56–58C

sequence containers
- see standard sequence containers

`set`
- constness of elements, 95C
- corrupting via element modification, 97C
- iterator invalidation in, see iterators, invalidation
- keys, modifying, 95–100C
- membership test, idiomatic, 199C

`set_difference`, 148C

`set_intersection`, 148C

`set_symmetric_difference`, 148C, 153C

`set_union`, 148C

`sgetc`, 127C

SGI
- hashed containers implementation, 114C
- iostreams implementation, 220C
- `slist` implementation, 218C
- STL web site, 94C, 207C, 217–220C
  - thread-safety definition at, 58C
  - URL for, 217C, 227C

shared memory, allocators and, 55–56C

`shared_ptr`, see Boost, `shared_ptr`
shrink to fit, see swap trick, the
side effects
    `accumulate` and, 160C
    `for_each` and, 160C
    in `operator++`, 45C
`size`
    vs. `capacity`, 66–67C
    vs. `empty`, 23–24C
`size_type`, 158C
`sizeof`, variations when applied to `string`, 69C
`skipws`, 126C
slicing problem, 21–22C, 164C
`slist`, 218C
small string optimization, 71C
smart pointers
    see also Boost, `shared_ptr`
    avoiding resource leaks with, 39C, 146C
    dereferencing function object for, 91C
    implicit conversions and, 146C
`sort`, 133–138C
    `qsort` vs. 203C
sorted range
    algorithms requiring, 146–150C
sorted `vector`s
    associative containers vs. 100–106C
    legacy APIs and, 76C
sorting
    algorithms for, 133–138C
    `auto_ptr`s, 41C
    consistency and, 149C
stability, in sorting, 135C
`stable_partition`, 133–138C

`stable_sort`, 133–138C
`stack`, as part of the STL, 2C
standard associative containers
    see also containers
    bidirectional iterators and, 5C
    comparison funcs for pointers, 88–91C
    definition of, 5C
    “hint” form of `insert`, 100C, 110C
    iterator invalidation in, see iterators, invalidation
    `key_comp` member function, 85C
    search complexity of, 6C
    sorted `vector` vs. 100–106C
    typical implementation, 52C, 190C
Standard for C++
    see C++ Standard, The
standard sequence containers
    see also containers
    definition of, 5C
    `erase`’s return value, 46C
    iterator invalidation in, see iterators, invalidation
    `push_back`, `back_inserter` and, 130C
    `push_front`, `front_inserter` and, 130C
Standard Template Library, see STL
Stepanov, Alexander, 201C
STL
    algorithms, vs. `string` member functions, 230C
    arrays and, 2C
    `bitset` and, 2C
    complexity guarantees in, 5–6C
    container adapters and, 2C
    containers, selecting among, 11–15C
    definition of, 2C
    documentation, on-line, 217C

    - extending, 2C
    - free implementations of, 217C, 220C
    - function call syntax in, 175C
    - implementations
        - compiler implementations vs. 3C
        - Dinkumware bug list for MSVC, 244C
        - replacing, 243C
    - member templates in, 239–240C
    - platform, see STL platform
    - `priority_queue` and, 2C
    - `queue` and, 2C
    - `stack` and, 2C
    - thread safety and, 58–62C
    - `valarray` and, 2C
    - web sites about, 217–223C
    - wide-character strings and, 4C
- STL platform
    - definition of, 3C
    - Microsoft's, remarks on, 239–244C
- STLport, 220–221C
    - debug mode, 185C, 216C
    - detecting invalidated iterators in, 221C
    - hashed containers at, 112C
    - URL for, 217C
- `strcmp`, 152C, 234C
    - internationalization issues and, 150C
    - `lexicographical_compare` and, 153C
- `strcmpi`, 154C
- streams, relation to `string`, 230C
- `stricmp`, 154C
- strict weak ordering, 94C
- `string`
    - allocators in, 69C

alternatives to, 65C
arrays vs. 63C–66C
as typedef for `basic_string`, 65C
`c_str` member function, 75C
comparisons
  case-insensitive, 150–154C, 235–237C
  using locales, 229–237C
cost of increasing capacity, 66C
disabling reference counting, 65C
embedded nulls and, 75C, 154C
growth factor, 66C
implementation variations, 68–73C
inheriting from, 37C
iterator invalidation in, see iterators, invalidation
iterators as pointers, 120C
legacy APIs and, 74–77C
mem funcs vs. algorithms, 230C
memory layout for, 75C
minimum allocation for, 72C
multithreading and, 64–65C
reference counting and, 64–65C
relation to `basic_string`, 4C, 210C, 229C
relation to streams, 230C
`reserve`, input iterators and, 131C
`resize` vs. `reserve`, 67C
shrink to fit, 78–79C
`size` vs. `capacity`, 66–67C
`size_type` typedef, 158C
`sizeof`, variations in, 69C
small, optimization for, 71C
summing lengths of, 158C
trimming extra capacity from, 77–79C
`vector` vs. 64C

    `vector<char>` vs. 65C
    whether reference-counted, 65C
    `wstring` and, 4C
`string_char_traits`, 211C
strings
    case-insensitive, 229–230C
    wide-character, see `wstring`
Stroustrup, Bjarne, 68C, 226C, 228C
    see also *C++ Programming Language, The*
structs, vs. classes for functor classes, 171C
summarizing ranges, 156–161C
Sutter, Herb, 65C, 226C, 227C, 228C
    see also *Exceptional C++*
swap trick, the, 77–79C

T

templates
    declaring inside functions, 188C
    explicit argument specification for, 122C, 163C, 234C
    instantiating with local classes, 189C
    member
        in the STL, 239–240C
        Microsoft's platforms and, 239–244C
    parameters, declared via `class` vs. `typename`, 7C
temporary objects, created via casts, 98C
thread safety, in containers, 58–62C
    see also multithreading
`tolower`, 151C
    as inverse of `toupper`, 235C
`toupper`
    as inverse of `tolower`, 235C
    cost of calling, 236–237C
traits classes, 113C, 211C, 230C

transactional semantics, 14C
`transform`, 129C, 186C
traversal order, in `multiset`, `multimap`, 87C
trees, red-black, 190C, 191C, 214C
typedefs
  `allocator::pointer`, 48C
  `allocator::reference`, 48C
  `argument_type`, 170C
  container `::size_type`, 158C
  container ::`value_type`, 36C
  `first_argument_type`, 170C
  for container and iterator types, 18–19C
  `mem_fun` and, 176C
  `mem_fun_ref` and, 176C
  `ptr_fun` and, 170C
  `result_type`, 170C
  `second_argument_type`, 170C
  `string` as, 65C
  `wstring` as, 65C
`typename`
  `class` vs. 7C
  dependent types and, 7–8C

U

`unary_function`, 170–172C
  pointer parameters and, 172C
  reference parameters and, 171C
undefined behavior
  accessing `v[0]` when `v` is empty, 74C
  applying some algorithms to ranges of unsorted values, 147C
  associative container comparison funcs yielding `true` for equal values, 92C
  attempting to modify objects defined to be `const`, 99C

  changing a `set` or `multiset` key, 97C
  deleting an object more than once, 64C
  deleting derived object via ptr-to-base with a nonvirtual destructor, 38C
  detecting via STLport's debug mode, 220–221C
  modifying components in `std`, 178C
  multithreading and, 59C
  practical meaning of, 3–4C
  side effects inside `accumulate`'s function object, 160C
  specifying uninitialized memory as destination range for algorithms, 132C
  using algorithms with inconsistent sort orders, 149C
  using the wrong form of `delete`, 64C
  when using invalidated iterator, 27C, 45C, 46C, 185C
underscores, in identifiers, 213C
`uninitialized_fill`, 52C
`uniq`, 148C
`unique`, 145C, 148C, 149C
  see also `remove`
`unique_copy`, 148C, 149C
`unsigned char`, use in `<cctype>` and `<ctype.h>`, 151C
`upper_bound`, 197–198C
  related functions vs. 192–201C
Urbano, Nancy L., see Perfect Woman
URLs
  for Austern's sample allocator, 228C
  for `auto_ptr` update page, 228C
  for Boost web site, 217C, 227C
  for Dinkumware web site, 243C
  for *Effective C++ CD* demo, 225C
  for *Effective C++ CD* errata list, 228C
  for *Effective C++* errata list, 228C
  for Josuttis' sample allocator, 227C
  for *More Effective C++* errata list, 228C

- for Persephone’s web site, 71C
- for purchasing The C++ Standard, 226C
- for SGI STL web site, 217C, 227C
- for STLport web site, 217C

`use_facet`, 234C
- cost of calling, 234C
- explicit template argument specification and, 234C

Usenet newsgroups, see newsgroups

V

`valarray`, as part of the STL, 2C

`value_type` typedef
- in containers, 36C, 108C
- in `iterator_traits`, 42C

`vector`
- see also `vector<bool>`, `vector<char>`
- arrays vs. 63C–66C
- contiguous memory for, 74C
- cost of increasing capacity, 66C
- growth factor, 66C
- iterator invalidation in, see iterators, invalidation
- iterators as pointers, 120C
- legacy APIs and, 74–77C
- `reserve`, input iterators and, 131C
- `resize` vs. `reserve`, 67C
- return type from `begin`, 75C
- shrink to fit, 78–79C
- `size` vs. `capacity`, 66–67C
- sorted
  - legacy APIs and, 76C
  - vs. associative containers, 100–106C
- `string` vs. 64C
- trimming extra capacity from, 77–79C

`vector::insert`, as member template, 240C
`vector<bool>` alternatives to, 81C
  legacy APIs and, 81C
  problems with, 79–82C
  proxy objects and, 80C
`vector<char>`, vs. `string`, 65C
virtual functions, `toupper` and, 236–237C
Visual Basic, 127C
Visual C++, see Microsoft's STL platforms
Vlissides, John, 226C
  see also *Design Patterns*
vocabulary, algorithms as, 186C

W

`wchar_t`, 4C
web sites
  see also URLs
  for STL-related resources, 217–223C
whitespace, `operator<<` and, 126C
wide-character strings, see `wstring`
`Widget`, use in this book, 9C
Wizard of Oz, *The*, 83C
Woman, Perfect, see Urbano, Nancy L.
wombat, 88C
workarounds
  for improperly declared iterator-related functions, 119C
  for Microsoft's STL platforms, 242–244C
  for missing member templates, 242C
  for `use_facet`, 234C
  function objects as, 204C
write-only code, avoiding, 206–208C
`wstring`
  as typedef for `basic_string`, 65C

    relation to `basic_string`, 4C
    `string` and, 4C
    this book and, 4C

Z

Zolman, Leor, 211C, 228C

```
extern int x;                              // object declaration
std::size_t numDigits(int number);         // function declaration
class Widget;                              // class declaration
template<typename T>                       // template declaration
class GraphNode;                           // (see Item 42 for info on
                                           // the use of "typename")
```

```
int x;                                            // object definition

std::size_t numDigits(int number)                 // function definition.
{                                                 // (This function returns
   std::size_t digitsSoFar = 1;                   // the number of digits
                                                  // in its parameter.)
   while ((number /= 10) != 0) ++digitsSoFar;

   return digitsSoFar;
}

class Widget {                                    // class definition
public:
   Widget();
   ~Widget();
   ...
};

template<typename T>                              // template definition
class GraphNode {
public:
   GraphNode();
   ~GraphNode();
   ...
};
```

```
class A {
public:
  A();                                    // default constructor
};

class B {
public:
  explicit B(int x = 0, bool b = true);   // default constructor; see below
};                                        // for info on "explicit"

 class C {
 public:
   explicit C(int x);                      // not a default constructor
 };
```

```
void doSomething(B bObject);    // a function taking an object of
                                // type B

B bObj1;                        // an object of type B

doSomething(bObj1);             // fine, passes a B to doSomething

B bObj2(28);                    // fine, creates a B from the int 28
                                // (the bool defaults to true)

doSomething(28);                // error! doSomething takes a B,
                                // not an int, and there is no
                                // implicit conversion from int to B

doSomething(B(28));             // fine, uses the B constructor to
                                // explicitly convert (i.e., cast) the
                                // int to a B for this call. (See
                                // Item 27 for info on casting.)
```

```
class Widget {
public:
  Widget();                                 // default constructor
  Widget(const Widget& rhs);                // copy constructor
  Widget& operator=(const Widget& rhs);     // copy assignment operator
  ...
};

Widget w1;                                  // invoke default constructor

Widget w2(w1);                              // invoke copy constructor

w1 = w2;                                    // invoke copy
                                            // assignment operator
```

```
bool hasAcceptableQuality(Widget w);
...
Widget aWidget;
if (hasAcceptableQuality(aWidget)) ...
```

```
int *p = 0;                   // p is a null pointer
std::cout << *p;              // dereferencing a null pointer
                              // yields undefined behavior
char name[] = "Darla";        // name is an array of size 6 (don’t
                              // forget the trailing null!)
char c = name[10];            // referring to an invalid array index
                              // yields undefined behavior
```

```
Widget *pw;                  // pw = ptr to Widget
class Airplane;
Airplane *pa;                // pa = ptr to Airplane
class GameCharacter;
GameCharacter *pgc;          // pgc = ptr to GameCharacter
```

```
class GamePlayer {
private:
   static const int NumTurns = 5;            // constant declaration
   int scores[NumTurns];                     // use of constant
   ...
};
```

```
const int GamePlayer::NumTurns;      // definition of NumTurns; see
                                     // below for why no value is given
```

```
class CostEstimate {
private:
   static const double FudgeFactor;          // declaration of static class
   ...                                       // constant; goes in header file
};

const double                                 // definition of static class
   CostEstimate::FudgeFactor = 1.35;         // constant; goes in impl. file
```

```
class GamePlayer {
private:
   enum { NumTurns = 5 };             // "the enum hack" — makes
                                      // NumTurns a symbolic name for 5

   int scores[NumTurns];              // fine

   ...

};
```

```
int a = 5, b = 0;

CALL_WITH_MAX(++a, b);                // a is incremented twice
CALL_WITH_MAX(++a, b+10);             // a is incremented once
```

```
template<typename T>                               // because we don't
inline void callWithMax(const T& a, const T& b)    // know what T is, we
{                                                  // pass by reference-to-
   f(a > b ? a : b);                               // const — see Item 20
}
```

```
char greeting[] = "Hello";
char *p = greeting;                     // non-const pointer,
                                        // non-const data
const char *p = greeting;               // non-const pointer,
                                        // const data
char * const p = greeting;              // const pointer,
                                        // non-const data
const char * const p = greeting;        // const pointer,
                                        // const data
```

```
void f1(const Widget *pw);          // f1 takes a pointer to a
                                    // constant Widget object
void f2(Widget const *pw);          // so does f2
```

```
std::vector<int> vec;
...
const std::vector<int>::iterator iter =       // iter acts like a T* const
   vec.begin();
*iter = 10;                                   // OK, changes what iter points to
++iter;                                       // error! iter is const

std::vector<int>::const_iterator cIter =      //cIter acts like a const T*
   vec.begin();
*cIter = 10;                                  // error! *cIter is const
++cIter;                                      // fine, changes cIter
```

```
class Rational { ... };
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

```
Rational a, b, c;
...
(a * b) = c;                    // invoke operator= on the
                                // result of a*b!
```

```
class TextBlock {
public:
   ...
   const char& operator[](std::size_t position) const      // operator[] for
   { return text[position]; }                              // const objects

   char& operator[](std::size_t position)                  // operator[] for
   { return text[position]; }                              // non-const objects

private:
    std::string text;
};
```

```
TextBlock tb("Hello");
std::cout << tb[0];                    // calls non-const
                                       // TextBlock::operator[]
const TextBlock ctb("World");
std::cout << ctb[0];                   // calls const TextBlock::operator[]
```

```
void print(const TextBlock& ctb)          // in this function, ctb is const
{
   std::cout << ctb[0];                   // calls const TextBlock::operator[]
   ...
}
```

```
std::cout << tb[0];            // fine — reading a
                               // non-const TextBlock

tb[0] = ’x’;                   // fine — writing a
                               // non-const TextBlock

std::cout << ctb[0];           // fine — reading a
                               // const TextBlock

ctb[0] = ’x’;                  // error! — writing a
                               // const TextBlock
```

```
class CTextBlock {
public:
   ...
   char& operator[](std::size_t position) const      // inappropriate (but bitwise
   { return pText[position]; }                        // const) declaration of
                                                      // operator[]
private:
   char *pText;
};
```

```
const CTextBlock cctb("Hello");          // declare constant object
char *pc = &cctb[0];                     // call the const operator[] to get a
                                         // pointer to cctb's data
*pc = 'J';                               // cctb now has the value "Jello"
```

```
class CTextBlock {
public:
   ...

   std::size_t length() const;

private:
   char *pText;
   std::size_t textLength;                // last calculated length of textblock
   bool lengthIsValid;                    // whether length is currently valid
};

std::size_t CTextBlock::length() const
{
   if (!lengthIsValid) {
      textLength = std::strlen(pText);    // error! can't assign to textLength
      lengthIsValid = true;               // and lengthIsValid in a const
   }                                      // member function

   return textLength;
}
```

```
class CTextBlock {
public:
  ...

  std::size_t length() const;

private:
  char *pText;

  mutable std::size_t textLength;         // these data members may
  mutable bool lengthIsValid;             // always be modified, even in
};                                        // const member functions

std::size_t CTextBlock::length() const
{
  if (!lengthIsValid) {
    textLength = std::strlen(pText);      // now fine
    lengthIsValid = true;                 // also fine
  }

  return textLength;
}
```

```
class TextBlock {
public:
  ...
  const char& operator[](std::size_t position) const
  {
    ...                                 // do bounds checking
    ...                                 // log access data
    ...                                 // verify data integrity
    return text[position];
  }
  char& operator[](std::size_t position)
  {
    ...                                 // do bounds checking
    ...                                 // log access data
    ...                                 // verify data integrity
    return text[position];
  }
private:
   std::string text;
};
```

```
class TextBlock {
public:

    ...
    const char& operator[](std::size_t position) const          // same as before
    {
        ...
        ...
        ...
        return text[position];
    }

    char& operator[](std::size_t position)                  // now just calls const op[]
    {
        return
            const_cast<char&>(                              // cast away const on
                                                            // op[]’s return type;
                static_cast<const TextBlock&>(*this)        // add const to *this’s type;
                    [position]                              // call const version of op[]
            );
    }

...

};
```

```
int x = 0;                                  // manual initialization of an int

const char * text = "A C-style string";     // manual initialization of a
                                            // pointer (see also Item 3)

double d;                                   // “initialization” by reading from
std::cin >> d;                              // an input stream
```

```
class PhoneNumber { ... };

class ABEntry {                              // ABEntry = “Address Book Entry”
public:
  ABEntry(const std::string& name, const std::string& address,
          const std::list<PhoneNumber>& phones);
private:
  std::string theName;
  std::string theAddress;
  std::list<PhoneNumber> thePhones;
  int numTimesConsulted;
};

ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)
{
  theName = name;                            // these are all assignments,
  theAddress = address;                      // not initializations
  thePhones = phones;
  numTimesConsulted = 0;
}
```

```
ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)
: theName(name),
  theAddress(address),                  // these are now all initializations
  thePhones(phones),
  numTimesConsulted(0)
{}                                      // the ctor body is now empty
```

```
ABEntry::ABEntry()
: theName(),                                    // call theName's default ctor;
  theAddress(),                                 // do the same for theAddress;
  thePhones(),                                  // and for thePhones;
  numTimesConsulted(0)                          // but explicitly initialize
{}                                              // numTimesConsulted to zero
```

```
class FileSystem {                  // from your library’s header file
public:
  ...
  std::size_t numDisks() const;     // one of many member functions
  ...
};

extern FileSystem tfs;              // declare object for clients to use
                                    // (“tfs” = “the file system” ); definition
                                    // is in some .cpp file in your library
```

```
class Directory {                           // created by library client
public:
    Directory( params );
   ...
};

Directory::Directory( params )
{
   ...
   std::size_t disks = tfs.numDisks();      // use the tfs object
   ...
}
```

```
class FileSystem { ... };              // as before

FileSystem& tfs()                      // this replaces the tfs object; it could be
{                                      // static in the FileSystem class

   static FileSystem fs;               // define and initialize a local static object
   return fs;                          // return a reference to it
}

class Directory { ... };               // as before

Directory::Directory( params )         // as before, except references to tfs are
{                                      // now to tfs()
   ...
   std::size_t disks = tfs().numDisks();
   ...
}

Directory& tempDir()                   // this replaces the tempDir object; it
{                                      // could be static in the Directory class

   static Directory td( params );      // define/initialize local static object
   return td;                          // return reference to it
}
```

```
class Empty {
public:
   Empty() { ... }                                // default constructor
   Empty(const Empty& rhs) { ... }                // copy constructor

   ~Empty() { ... }                               // destructor — see below
                                                  // for whether it’s virtual

   Empty& operator=(const Empty& rhs) { ... }     // copy assignment operator
};
```

```
Empty e1;                       // default constructor;
                                // destructor
Empty e2(e1);                   // copy constructor
e2 = e1;                        // copy assignment operator
```

```
template<typename T>
class NamedObject {
public:
   NamedObject(const char *name, const T& value);
   NamedObject(const std::string& name, const T& value);

   ...

private:
   std::string nameValue;
   T objectValue;
};
```

```
NamedObject<int> no1("Smallest Prime Number", 2);
NamedObject<int> no2(no1);                          // calls copy constructor
```

```
template<typename T>
class NamedObject {
public:
  // this ctor no longer takes a const name, because nameValue
  // is now a reference-to-non-const string. The char* constructor
  // is gone, because we must have a string to refer to.
  NamedObject(std::string& name, const T& value);

  ...                                        // as above, assume no
                                             // operator= is declared
private:
  std::string& nameValue;                    // this is now a reference
  const T objectValue;                       // this is now const
};
```

```
std::string newDog("Persephone");
std::string oldDog("Satch");

NamedObject<int> p(newDog, 2);        // when I originally wrote this, our
                                      // dog Persephone was about to
                                      // have her second birthday

NamedObject<int> s(oldDog, 36);       // the family dog Satch (from my
                                      // childhood) would be 36 if she
                                      // were still alive

p = s;                                // what should happen to
                                      // the data members in p?
```

```
HomeForSale h1;
HomeForSale h2;

HomeForSale h3(h1);              // attempt to copy h1 — should
                                 // not compile!

h1 = h2;                         // attempt to copy h2 — should
                                 // not compile!
```

```
class HomeForSale {
public:
   ...
private:
   ...
   HomeForSale(const HomeForSale&);                    // declarations only
   HomeForSale& operator=(const HomeForSale&);
};
```

```
class Uncopyable {
protected:                                          // allow construction
   Uncopyable() {}                                  // and destruction of
   ~Uncopyable() {}                                 // derived objects...

private:
   Uncopyable(const Uncopyable&);                   // ...but prevent copying
   Uncopyable& operator=(const Uncopyable&);
};
```

```
class HomeForSale: private Uncopyable {          // class no longer
   ...                                           // declares copy ctor or
};                                               // copy assign. operator
```

```
class TimeKeeper {
public:
   TimeKeeper();
   ~TimeKeeper();
   ...
};

class AtomicClock: public TimeKeeper { ... };

class WaterClock: public TimeKeeper { ... };

class WristWatch: public TimeKeeper { ... };
```

```
TimeKeeper* getTimeKeeper();        // returns a pointer to a dynamic-
                                    // ally allocated object of a class
                                    // derived from TimeKeeper
```

```
TimeKeeper *ptk = getTimeKeeper();   // get dynamically allocated object
                                     // from TimeKeeper hierarchy
...                                  // use it
delete ptk;                          // release it to avoid resource leak
```

```
class TimeKeeper {
public:
   TimeKeeper();
   virtual ~TimeKeeper();
   ...
};

TimeKeeper *ptk = getTimeKeeper();

...

delete ptk;                          // now behaves correctly
```

```
class Point {                                  // a 2D point
public:
   Point(int xCoord, int yCoord);
   ~Point();

private:
   int x, y;
};
```

```
class SpecialString: public std::string {      // bad idea! std::string has a
  ...                                          // non-virtual destructor
};
```

```
SpecialString *pss =new SpecialString("Impending Doom");
std::string *ps;
...
ps = pss;                            // SpecialString* ⇒  std::string*
...
delete ps;                           // undefined! In practice,
                                     // *ps’s SpecialString resources
                                     // will be leaked, because the
                                     // SpecialString destructor won’t
                                     // be called.
```

```
class AWOV {                        // AWOV = "Abstract w/o Virtuals"
public:
   virtual ~AWOV() = 0;             // declare pure virtual destructor
};
```

```
class Widget {
public:
  ...
  ~Widget() { ... }                    // assume this might emit an exception
};

void doSomething()
{
  std::vector<Widget> v;
  ...
}                                      // v is automatically destroyed here
```

```
class DBConnection {
public:
   ...
   static DBConnection create();         // function to return
                                         // DBConnection objects; params
                                         // omitted for simplicity

   void close();                         // close connection; throw an
};                                       // exception if closing fails
```

```
class DBConn {                          // class to manage DBConnection
public:                                 // objects
   ...
   ~DBConn()                            // make sure database connections
   {                                    // are always closed
      db.close();
   }

private:
   DBConnection db;
};
```

```
{                                       // open a block
  DBConn dbc(DBConnection::create());   // create DBConnection object
                                        // and turn it over to a DBConn
                                        // object to manage
  ...                                   // use the DBConnection object
                                        // via the DBConn interface
}                                       // at end of block, the DBConn
                                        // object is destroyed, thus
                                        // automatically calling close on
                                        // the DBConnection object
```

```
DBConn::~DBConn()
{
   try { db.close(); }
   catch (...) {
      make log entry that the call to close failed;
      std::abort();
   }
}
```

```
DBConn::~DBConn()
{
   try { db.close(); }
   catch (...) {
      make log entry that the call to close failed;
   }
}
```

```
class DBConn {
public:
   ...
   void close()                                         // new function for
   {                                                    // client use
      db.close();
      closed = true;
   }

   ~DBConn()
   {
      if (!closed) {
         try {                                          // close the connection
            db.close();                                 // if the client didn't
         }
         catch (...) {                                  // if closing fails,
            make log entry that call to close failed;   // note that and
            ...                                         // terminate or swallow
         }
      }
   }

private:
   DBConnection db;
   bool closed;
};
```

```
class Transaction {                                  // base class for all
public:                                              // transactions
  Transaction();

  virtual void logTransaction() const = 0;           // make type-dependent
                                                     // log entry
  ...

};

Transaction::Transaction()                           // implementation of
{                                                    // base class ctor
  ...
  logTransaction();                                  // as final action, log this
}                                                    // transaction

class BuyTransaction: public Transaction {           // derived class
public:
  virtual void logTransaction() const;               // how to log trans-
                                                     // actions of this type
  ...

};

class SellTransaction: public Transaction {          // derived class
public:
  virtual void logTransaction() const;               // how to log trans-
                                                     // actions of this type
  ...

};
```

```
class Transaction {
public:
   Transaction()
   { init(); }                                   // call to non-virtual...

   virtual void logTransaction() const = 0;
   ...

private:
   void init()
   {
      ...
      logTransaction();                          // ...that calls a virtual!
   }
};
```

```
class Transaction {
public:
  explicit Transaction(const std::string& logInfo);

  void logTransaction(const std::string& logInfo) const;      // now a non-
                                                               // virtual func
  ...
};

Transaction::Transaction(const std::string& logInfo)
{
  ...
  logTransaction(logInfo);                                     // now a non-
}                                                              // virtual call

class BuyTransaction: public Transaction {
public:
  BuyTransaction( parameters )
  : Transaction(createLogString( parameters ))                 // pass log info
  { ... }                                                      // to base class
  ...                                                          // constructor

private:
  static std::string createLogString( parameters );
};
```

```
   class Widget {
   public:
     ...
   Widget& operator=(const Widget& rhs)      // return type is a reference to
   {                                         // the current class
     ...
     return *this;                           // return the left-hand object
   }
   ...
};
```

```
class Widget {
public:
  ...
  Widget& operator+=(const Widget& rhs)   // the convention applies to
  {                                        // +=, -=, *=, etc.
    ...
    return *this;
  }

  Widget& operator=(int rhs)               // it applies even if the
  {                                        // operator's parameter type
    ...                                    // is unconventional
    return *this;
  }
  ...
};
```

```
class Widget { ... };
Widget w;
...
w = w;                                  // assignment to self
```

```
class Base { ... };
class Derived: public Base { ... };
void doSomething(const Base& rb,        // rb and *pd might actually be
                       Derived* pd);    // the same object
```

```
class Bitmap { ... };
class Widget {
   ...
private:
   Bitmap *pb;                                    // ptr to a heap-allocated object
};
```

```
Widget&
Widget::operator=(const Widget& rhs)  // unsafe impl. of operator=
{
  delete pb;                          // stop using current bitmap
  pb = new Bitmap(*rhs.pb);           // start using a copy of rhs's bitmap

  return *this;                       // see Item 10
}
```

```
Widget& Widget::operator=(const Widget& rhs)
{
   if (this == &rhs) return *this;          // identity test: if a self-assignment,
                                            // do nothing
   delete pb;
   pb = new Bitmap(*rhs.pb);

   return *this;
}
```

```
Widget& Widget::operator=(const Widget& rhs)
{
  Bitmap *pOrig = pb;                // remember original pb
  pb = new Bitmap(*rhs.pb);          // make pb point to a copy of *pb
  delete pOrig;                      // delete the original pb

  return *this;
}
```

```
class Widget {
  ...
  void swap(Widget& rhs);              // exchange *this's and rhs's data;
  ...                                  // see Item 29 for details
};

Widget& Widget::operator=(const Widget& rhs)
{
  Widget temp(rhs);                    // make a copy of rhs's data

  swap(temp);                          // swap *this's data with the copy's

  return *this;
}
```

```
Widget& Widget::operator=(Widget rhs)   // rhs is a copy of the object
{                                       // passed in — note pass by val

  swap(rhs);                            // swap *this's data with
                                        // the copy's

  return *this;
}
```

```
void logCall(const std::string& funcName);          // make a log entry

class Customer {
public:
  ...
  Customer(const Customer& rhs);
  Customer& operator=(const Customer& rhs);
  ...

private:
  std::string name;
};

Customer::Customer(const Customer& rhs)
: name(rhs.name)                                    // copy rhs's data
{
  logCall("Customer copy constructor");
}

Customer& Customer::operator=(const Customer& rhs)
{
  logCall("Customer copy assignment operator");

  name = rhs.name;                                  // copy rhs's data

  return *this;                                     // see Item 10
}
```

```
class Date { ... };                              // for dates in time

class Customer {
public:
   ...                                           // as before

private:
   std::string name;
   Date lastTransaction;
};
```

```
class PriorityCustomer: public Customer {              // a derived class
public:
   ...
   PriorityCustomer(const PriorityCustomer& rhs);
   PriorityCustomer& operator=(const PriorityCustomer& rhs);
   ...

private:
   int priority;
};

PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
: priority(rhs.priority)
{
   logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
   logCall("PriorityCustomer copy assignment operator");

   priority = rhs.priority;

   return *this;
}
```

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
:  Customer(rhs),                         // invoke base class copy ctor
   priority(rhs.priority)
{
   logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
   logCall("PriorityCustomer copy assignment operator");

   Customer::operator=(rhs);              // assign base class parts
   priority = rhs.priority;

   return *this;
}
```

```
Investment* createInvestment();    // return ptr to dynamically allocated
                                   // object in the Investment hierarchy;
                                   // the caller must delete it
                                   // (parameters omitted for simplicity)
```

```
void f()
{
   Investment *pInv = createInvestment();          // call factory function
   ...                                             // use pInv
   delete pInv;                                    // release object
}
```

```
void f()
{
   std::auto_ptr<Investment> pInv(createInvestment());    // call factory
                                                          // function
   ...                                                    // use pInv as
                                                          // before
}                                                         // automatically
                                                          // delete pInv via
                                                          // auto_ptr's dtor
```

```
std::auto_ptr<Investment>                  // pInv1 points to the
   pInv1(createInvestment());              // object returned from
                                           // createInvestment

std::auto_ptr<Investment> pInv2(pInv1);    // pInv2 now points to the
                                           // object; pInv1 is now null

pInv1 = pInv2;                             // now pInv1 points to the
                                           // object, and pInv2 is null
```

```
void f()
{
  ...
  std::tr1::shared_ptr<Investment>
    pInv(createInvestment());                // call factory function

  ...                                        // use pInv as before

}                                            // automatically delete
                                             // pInv via shared_ptr's dtor
```

```
void f()
{
  ...
  std::tr1::shared_ptr<Investment>          // pInv1 points to the
    pInv1(createInvestment());               // object returned from
                                             // createInvestment

  std::tr1::shared_ptr<Investment>          // both pInv1 and pInv2 now
    pInv2(pInv1);                            // point to the object

  pInv1 = pInv2;                             // ditto — nothing has
                                             // changed

  ...
}                                            // pInv1 and pInv2 are
                                             // destroyed, and the
                                             // object they point to is
                                             // automatically deleted
```

```
std::auto_ptr<std::string>                      // bad idea! the wrong
  aps(new std::string[10]);                     // delete form will be used

std::tr1::shared_ptr<int> spi(new int[1024]);   // same problem
```

```
void lock(Mutex *pm);                 // lock mutex pointed to by pm
void unlock(Mutex *pm);               // unlock the mutex
```

```
class Lock {
public:
   explicit Lock(Mutex *pm)
   : mutexPtr(pm)
   { lock(mutexPtr); }                               // acquire resource

   ~Lock() { unlock(mutexPtr); }                     // release resource

private:
   Mutex *mutexPtr;
};
```

```
Mutex m;                 // define the mutex you need to use
...
{                        // create block to define critical section
   Lock ml(&m);          // lock the mutex
   ...                   // perform critical section operations
}                        // automatically unlock mutex at end
                         // of block
```

```
Lock ml1(&m);                   // lock m
Lock ml2(ml1);                  // copy ml1 to ml2 — what should
                                // happen here?
```

```
class Lock: private Uncopyable {          // prohibit copying — see
public:                                   // Item 6

  ...                                     // as before

};
```

```
class Lock {
public:
  explicit Lock(Mutex *pm)           // init shared_ptr with the Mutex
  : mutexPtr(pm, unlock)             // to point to and the unlock func
  {                                  // as the deleter

    lock(mutexPtr.get());            // see Item 15 for info on "get"
  }

private:
  std::tr1::shared_ptr<Mutex> mutexPtr;      // use shared_ptr
};                                           // instead of raw pointer
```

```
int daysHeld(const Investment *pi);          // return number of days
                                             // investment has been held
```

```
int days = daysHeld(pInv.get());                    // fine, passes the raw pointer
                                                    // in pInv to daysHeld
```

```
class Investment {                          // root class for a hierarchy
public:                                     // of investment types
  bool isTaxFree() const;
  ...
};

Investment* createInvestment();                        // factory function

std::tr1::shared_ptr<Investment>                       // have tr1::shared_ptr
  pi1(createInvestment());                             // manage a resource

bool taxable1 = !(pi1->isTaxFree());                   // access resource
                                                       // via operator->
...

std::auto_ptr<Investment> pi2(createInvestment());     // have auto_ptr
                                                       // manage a
                                                       // resource

bool taxable2 = !((*pi2).isTaxFree());                 // access resource
                                                       // via operator*
...
```

```
FontHandle getFont();                  // from C API — params omitted
                                       // for simplicity

void releaseFont(FontHandle fh);       // from the same C API

class Font {                           // RAII class
public:
  explicit Font(FontHandle fh)         // acquire resource;
  : f(fh)                              // use pass-by-value, because the
  {}                                   // C API does

  ~Font() { releaseFont(f ); }         // release resource

  ...                                  // handle copying (see Item 14)

private:
  FontHandle f;                        // the raw font resource
};
```

```
class Font {
public:
   ...
   FontHandle get() const { return f; }      // explicit conversion function
   ...
};
```

```
void changeFontSize(FontHandle f, int newSize);     // from the C API

Font f(getFont());
int newFontSize;

...

changeFontSize(f.get(), newFontSize);               // explicitly convert
                                                    // Font to FontHandle
```

```
class Font {
public:
    ...
    operator FontHandle() const               // implicit conversion function
    { return f; }
    ...
};
```

```
Font f(getFont());
int newFontSize;
...
changeFontSize(f, newFontSize);          // implicitly convert Font
                                         // to FontHandle
```

```
Font f1(getFont());

...

FontHandle f2 = f1;                 // oops! meant to copy a Font
                                    // object, but instead implicitly
                                    // converted f1 into its underlying
                                    // FontHandle, then copied that
```

```
std::string *stringPtr1 = new std::string;
std::string *stringPtr2 = new std::string[100];
...
delete stringPtr1;                    // delete an object
delete [] stringPtr2;                 // delete an array of objects
```

```
typedef std::string AddressLines[4];        // a person’s address has 4 lines,
                                            // each of which is a string
```

```
std::string *pal = new AddressLines;    // note that “new AddressLines”
                                        // returns a string*, just like
                                        // “new string[4]” would
```

```
std::tr1::shared_ptr<Widget> pw(new Widget);    // store newed object
                                                // in a smart pointer in a
                                                // standalone statement
processWidget(pw, priority());                  // this call won't leak
```

```
struct Day {                 struct Month {                 struct Year {
  explicit Day(int d)          explicit Month(int m)          explicit Year(int y)
  : val(d) {}                  : val(m) {}                    : val(y){}

  int val;                     int val;                       int val;
};                           };                             };

class Date {
public:
  Date(const Month& m, const Day& d, const Year& y);
  ...
};

Date d(30, 3, 1995);                          // error! wrong types

Date d(Day(30), Month(3), Year(1995));        // error! wrong types

Date d(Month(3), Day(30), Year(1995));        // okay, types are correct
```

```
class Month {
public:
   static Month Jan() { return Month(1); }     // functions returning all valid
   static Month Feb() { return Month(2); }     // Month values; see below for
   ...                                         // why these are functions, not
   static Month Dec() { return Month(12); }    // objects

   ...                                         // other member functions

private:
   explicit Month(int m);                      // prevent creation of new
                                               // Month values

   ...                                         // month-specific data
};

Date d(Month::Mar(), Day(30), Year(1995));
```

```
Investment* createInvestment();     // from Item 13; parameters omitted
                                    // for simplicity
```

```
std::tr1::shared_ptr<Investment>         // attempt to create a null
   pInv(0, getRidOfInvestment);          // shared_ptr with a custom deleter;
                                         // this won't compile
```

```
std::tr1::shared_ptr<Investment>       // create a null shared_ptr with
  pInv( static_cast<Investment*>(0),   // getRidOfInvestment as its
        getRidOfInvestment);           // deleter; see Item 27 for info on
                                       // static_cast
```

```
std::tr1::shared_ptr<Investment> createInvestment()
{
  std::tr1::shared_ptr<Investment> retVal(static_cast<Investment*>(0),
                                          getRidOfInvestment);

  retVal = ... ;                         // make retVal point to the
                                         // correct object

  return retVal;
}
```

```
std::tr1::shared_ptr<Investment> createInvestment()
{
   return std::tr1::shared_ptr<Investment>(new Stock);
}
```

```
class Person {
public:
   Person();                            // parameters omitted for simplicity
   virtual ~Person();                   // see Item 7 for why this is virtual
   ...

private:
   std::string name;
   std::string address;
};

class Student: public Person {
public:
   Student();                           // parameters again omitted
   virtual ~Student();
   ...

private:
   std::string schoolName;
   std::string schoolAddress;
};
```

```
bool validateStudent(Student s);           // function taking a Student
                                           // by value
Student plato;                             // Plato studied under Socrates
bool platoIsOK = validateStudent(plato);   // call the function
```

```
class Window {
public:
   ...
   std::string name() const;                    // return name of window
   virtual void display() const;                // draw window and contents
};

class WindowWithScrollBars: public Window {
public:
   ...
   virtual void display() const;
};
```

```
void printNameAndDisplay(Window w)          // incorrect! parameter
{                                           // may be sliced!
   std::cout << w.name();
   w.display();
}
```

```
void printNameAndDisplay(const Window& w)     // fine, parameter won't
{                                             // be sliced
   std::cout << w.name();
   w.display();
}
```

```
class Rational {
public:
  Rational(int numerator = 0,                // see Item 24 for why this
           int denominator = 1);             // ctor isn't declared explicit

  ...

private:
  int n, d;                                  // numerator and denominator

friend
  const Rational                             // see Item 3 for why the
    operator*(const Rational& lhs,           // return type is const
              const Rational& rhs);
};
```

```
const Rational& operator*(const Rational& lhs,        // warning! bad code!
                          const Rational& rhs)
{
  Rational result(lhs.n * rhs.n, lhs.d * rhs.d);
  return result;
}
```

```
const Rational& operator*(const Rational& lhs,      // warning! more bad
                          const Rational& rhs)      // code!
{
  Rational *result = new Rational(lhs.n * rhs.n, lhs.d * rhs.d);
  return *result;
}
```

```
Rational w, x, y, z;
w = x * y * z;                    // same as operator*(operator*(x, y), z)
```

```
const Rational& operator*(const Rational& lhs,      // warning! yet more
                          const Rational& rhs)      // bad code!
{
  static Rational result;                     // static object to which a
                                              // reference will be returned

  result = ... ;                              // multiply lhs by rhs and put the
                                              // product inside result

  return result;
}
```

```
bool operator==(const Rational& lhs,            // an operator==
                const Rational& rhs);           // for Rationals

Rational a, b, c, d;

...

if ((a * b) == (c * d))  {

   do whatever's appropriate when the products are equal;

} else  {

   do whatever's appropriate when they're not;

}
```

```
inline const Rational operator*(const Rational& lhs, const Rational& rhs)
{
   return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

```
class AccessLevels {
public:
  ...
  int getReadOnly() const          { return readOnly; }

  void setReadWrite(int value)     { readWrite = value; }
  int getReadWrite() const         { return readWrite; }

  void setWriteOnly(int value)     { writeOnly = value; }

private:
  int noAccess;                             // no access to this int

  int readOnly;                             // read-only access to this int

  int readWrite;                            // read-write access to this int

  int writeOnly;                            // write-only access to this int
};
```

```
class SpeedDataCollection {
   ...
public:
   void addValue(int speed);              // add a new data value
   double averageSoFar() const;           // return average speed
   ...
};
```

```
class WebBrowser {
public:
  ...
  void clearEverything();                      // calls clearCache, clearHistory,
                                               // and removeCookies

  ...
};
```

```
void clearBrowser(WebBrowser& wb)
{
   wb.clearCache();
   wb.clearHistory();
   wb.removeCookies();
}
```

```
namespace WebBrowserStuff {
   class WebBrowser { ... };
   void clearBrowser(WebBrowser& wb);
   ...
}
```

```
// header "webbrowser.h" — header for class WebBrowser itself
// as well as "core" WebBrowser-related functionality
namespace WebBrowserStuff {

  class WebBrowser { ... };

    ...                                 // "core" related functionality, e.g.
                                        // non-member functions almost
                                        // all clients need
}

// header "webbrowserbookmarks.h"
namespace WebBrowserStuff {
  ...                                   // bookmark-related convenience
}                                       // functions

// header "webbrowsercookies.h"
namespace WebBrowserStuff {
  ...                                   // cookie-related convenience
}                                       // functions

...
```

```
class Rational {
public:
  Rational(int numerator = 0,          // ctor is deliberately not explicit;
           int denominator = 1);       // allows implicit int-to-Rational
                                       // conversions

  int numerator() const;               // accessors for numerator and
  int denominator() const;             // denominator — see Item 22

private:
  ...
};
```

```
class Rational {
public:
   ...
   const Rational operator*(const Rational& rhs) const;
};
```

```
Rational oneEighth(1, 8);
Rational oneHalf(1, 2);
Rational result = oneHalf * oneEighth;          // fine
result = result * oneEighth;                    // fine
```

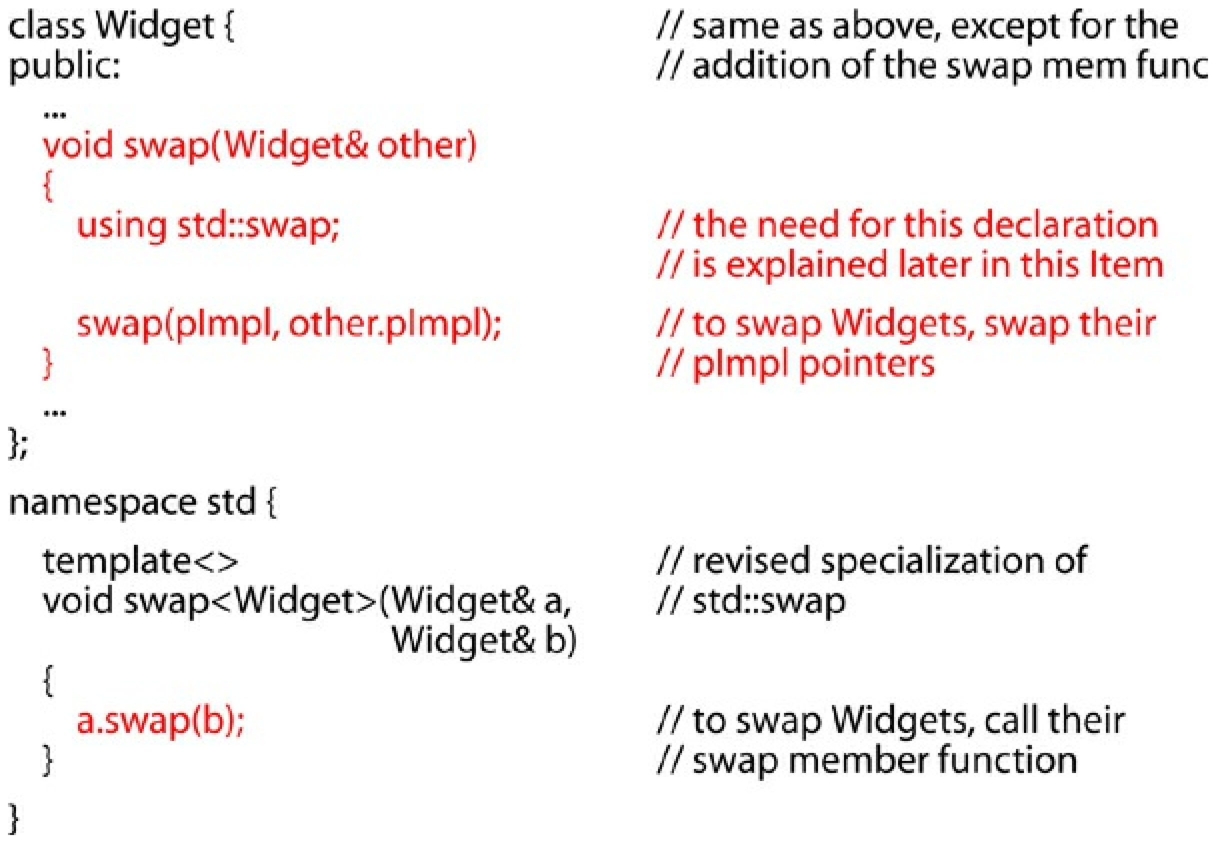
```
const Rational temp(2);          // create a temporary
                                 // Rational object from 2
result = oneHalf * temp;         // same as oneHalf.operator*(temp);
result = oneHalf * 2;            // error! (with explicit ctor);
                                 // can’t convert 2 to Rational
result = 2 * oneHalf;            // same error, same problem
result = oneHalf * 2;                // fine (with non-explicit ctor)
result = 2 * oneHalf;                // error! (even with non-explicit ctor)
```

```
class Rational {
  ...                                                  // contains no operator*
};

const Rational operator*(const Rational& lhs,          // now a non-member
                         const Rational& rhs)          // function
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}

Rational oneFourth(1, 4);
Rational result;

result = oneFourth * 2;                                // fine
result = 2 * oneFourth;                                // hooray, it works!
```

```
namespace std {

   template<typename T>                 // typical implementation of std::swap;
   void swap(T& a, T& b)                // swaps a's and b's values
   {
      T temp(a);
      a = b;
      b = temp;
   }

}
```

```
class WidgetImpl {                          // class for Widget data;
public:                                     // details are unimportant
  ...

private:
  int a, b, c;                              // possibly lots of data —
  std::vector<double> v;                    // expensive to copy!
  ...
};
class Widget {                              // class using the pimpl idiom
public:
  Widget(const Widget& rhs);

  Widget& operator=(const Widget& rhs)      // to copy a Widget, copy its
  {                                         // WidgetImpl object. For
    ...                                     // details on implementing
    *pImpl = *(rhs.pImpl);                  // operator= in general,
    ...                                     // see Items 10, 11, and 12.
  }
  ...

private:
  WidgetImpl *pImpl;                        // ptr to object with this
};                                          // Widget's data
```

```
namespace std {
  template<>                          // this is a specialized version
  void swap<Widget>(Widget& a,        // of std::swap for when T is
                    Widget& b)        // Widget
  {
    swap(a.pImpl, b.pImpl);           // to swap Widgets, swap their
  }                                   // pImpl pointers; this won't
compile
}
```



```
class Widget {                                 // same as above, except for the
public:                                        // addition of the swap mem func
  ...
  void swap(Widget& other)
  {
    using std::swap;                           // the need for this declaration
                                               // is explained later in this Item

    swap(pImpl, other.pImpl);                  // to swap Widgets, swap their
  }                                            // pImpl pointers
  ...
};

namespace std {

  template<>                                   // revised specialization of
  void swap<Widget>(Widget& a,                 // std::swap
                    Widget& b)
  {
    a.swap(b);                                 // to swap Widgets, call their
  }                                            // swap member function

}
```

```
namespace std {

  template<typename T>
  void swap<Widget<T> >(Widget<T>& a,          // error! illegal code!
                        Widget<T>& b)
  { a.swap(b); }

}
```

```
namespace std {

   template<typename T>                    // an overloading of std::swap
   void swap(Widget<T>& a,                 // (note the lack of "<...>" after
             Widget<T>& b)                 // "swap"), but see below for
   { a.swap(b); }                          // why this isn't valid code

}
```

```
namespace WidgetStuff {
   ...                                      // templatized WidgetImpl, etc.

   template<typename T>                     // as before, including the swap
   class Widget { ... };                    // member function

   ...

   template<typename T>                     // non-member swap function;
   void swap(Widget<T>& a,                  // not part of the std namespace
             Widget<T>& b)
   {
     a.swap(b);
   }
}
```

```
template<typename T>
void doSomething(T& obj1, T& obj2)
{
   using std::swap;                    // make std::swap available in this function
   ...
   swap(obj1, obj2);                   // call the best swap for objects of type T
   ...
}
```

```
// this function defines the variable "encrypted" too soon
std::string encryptPassword(const std::string& password)
{
   using namespace std;

   string encrypted;

   if (password.length() < MinimumPasswordLength) {
       throw logic_error("Password is too short");
   }

   ...                                      // do whatever is necessary to place an
                                            // encrypted version of password in encrypted

   return encrypted;
}
```

```
// this function postpones encrypted's definition until it's truly necessary
std::string encryptPassword(const std::string& password)
{
  using namespace std;

  if (password.length() < MinimumPasswordLength) {
     throw logic_error("Password is too short");
  }

  string encrypted;

  ...                          // do whatever is necessary to place an
                               // encrypted version of password in encrypted

  return encrypted;
}
```

```
// this function postpones encrypted's definition until
// it's necessary, but it's still needlessly inefficient
std::string encryptPassword(const std::string& password)
{
  ...                                    // import std and check length as above

  string encrypted;                      // default-construct encrypted
  encrypted = password;                  // assign to encrypted

  encrypt(encrypted);
  return encrypted;
}
```

```
// finally, the best way to define and initialize encrypted
std::string encryptPassword(const std::string& password)
{
   ...                                    // import std and check length

   string encrypted(password);            // define and initialize via copy
                                          // constructor

   encrypt(encrypted);
   return encrypted;
}
```

```
// Approach A: define outside loop
Widget w;
for (int i = 0; i < n; ++i) {
   w = some value dependent on i;
i);
   ...
}
```

```
// Approach B: define inside loop

for (int i = 0; i < n; ++i) {
   Widget w(some value dependent on

   ...
}
```

```
class Widget {
public:
   explicit Widget(int size);
   ...
};

void doSomeWork(const Widget& w);

doSomeWork(Widget(15));                      // create Widget from int
                                             // with function-style cast

doSomeWork(static_cast<Widget>(15));         // create Widget from int
                                             // with C++-style cast
```

```
int x, y;
...
double d = static_cast<double>(x)/y;          // divide x by y, but use
                                              // floating point division
```

```
class Base { ... };
class Derived: public Base { ... };
Derived d;
Base *pb = &d;                                   // implicitly convert Derived* ⇒  Base*
```

```
class Window {                                    // base class
public:
  virtual void onResize() { ... }                 // base onResize impl
  ...
};

class SpecialWindow: public Window {              // derived class
public:
  virtual void onResize() {                       // derived onResize impl;
    static_cast<Window>(*this).onResize();        // cast *this to Window,
                                                  // then call its onResize;
                                                  // this doesn't work!

    ...                                           // do SpecialWindow-
  }                                               // specific stuff

  ...

};
```

```
class SpecialWindow: public Window {
public:
  virtual void onResize() {
    Window::onResize();                         // call Window::onResize
    ...                                         // on *this
  }

  ...

};
```

```
class Window { ... };

class SpecialWindow: public Window {
public:
  void blink();
  ...
};

typedef                                          // see Item 13 for info
  std::vector<std::tr1::shared_ptr<Window> > VPW;  // on tr1::shared_ptr

VPW winPtrs;

...

for (VPW::iterator iter = winPtrs.begin();       // undesirable code:
     iter != winPtrs.end();                       // uses dynamic_cast
     ++iter) {
  if (SpecialWindow *psw = dynamic_cast<SpecialWindow*>(iter->get()))
     psw->blink();
}
```

```
typedef std::vector<std::tr1::shared_ptr<SpecialWindow> > VPSW;
VPSW winPtrs;
...
for (VPSW::iterator iter = winPtrs.begin();          // better code: uses
     iter != winPtrs.end();                          // no dynamic_cast
     ++iter)
  (*iter)->blink();
```

```
class Window {
public:
  virtual void blink() {}                        // default impl is no-op;
  ...                                            // see Item 34 for why
};                                               // a default impl may be
                                                 // a bad idea

class SpecialWindow: public Window {
public:
  virtual void blink() { ... }                   // in this class, blink
  ...                                            // does something
};

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;

VPW winPtrs;                                     // container holds
                                                 // (ptrs to) all possible
...                                              // Window types

for (VPW::iterator iter = winPtrs.begin();
     iter != winPtrs.end();
     ++iter)                                     // note lack of
  (*iter)->blink();                              // dynamic_cast
```

```
class Window { ... };
...                                              // derived classes are defined here
typedef std::vector<std::tr1::shared_ptr<Window> > VPW;
VPW winPtrs;
...
for (VPW::iterator iter = winPtrs.begin(); iter != winPtrs.end(); ++iter)
{
   if (SpecialWindow1 *psw1 =
         dynamic_cast<SpecialWindow1*>(iter->get())) { ... }
   else if (SpecialWindow2 *psw2 =
              dynamic_cast<SpecialWindow2*>(iter->get())) { ... }
   else if (SpecialWindow3 *psw3 =
              dynamic_cast<SpecialWindow3*>(iter->get())) { ... }
   ...
}
```

```
class Point {                                  // class for representing points
public:
  Point(int x, int y);
  ...
  void setX(int newVal);
  void setY(int newVal);
  ...
};
struct RectData {                              // Point data for a Rectangle
  Point ulhc;                                  // ulhc = “ upper left-hand corner”
  Point lrhc;                                  // lrhc = “ lower right-hand corner”
};
class Rectangle {
  ...
private:
  std::tr1::shared_ptr<RectData> pData;                  // see Item 13 for info on
};                                                       // tr1::shared_ptr
```

```
class Rectangle {
public:
   ...
   Point& upperLeft() const { return pData->ulhc; }
   Point& lowerRight() const { return pData->lrhc; }
   ...
};
```

```
Point coord1(0, 0);
Point coord2(100, 100);
const Rectangle rec(coord1, coord2);        // rec is a const rectangle from
                                            // (0, 0) to (100, 100)
rec.upperLeft().setX(50);                   // now rec goes from
                                            // (50, 0) to (100, 100)!
```

```
class Rectangle {
public:
   ...
   const Point& upperLeft() const { return pData->ulhc; }
   const Point& lowerRight() const { return pData->lrhc; }
   ...
};
```

```
class GUIObject { ... };

const Rectangle                                  // returns a rectangle by
  boundingBox(const GUIObject& obj);             // value; see Item 3 for why
                                                 //  return type is const
```

```
GUIObject *pgo;                              // make pgo point to
...                                          // some GUIObject

const Point *pUpperLeft =                    // get a ptr to the upper
   &(boundingBox(*pgo).upperLeft());         // left point of its
                                             // bounding box
```

```
class PrettyMenu {
public:
  ...
  void changeBackground(std::istream& imgSrc);     // change background
  ...                                              // image
private:
   Mutex mutex;                        // mutex for this object

  Image *bgImage;                      // current background image
  int imageChanges;                    // # of times image has been changed
};
```

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  lock(&mutex);                      // acquire mutex (as in Item 14)

  delete bgImage;                    // get rid of old background
  ++imageChanges;                    // update image change count
  bgImage = new Image(imgSrc);       // install new background

  unlock(&mutex);                    // release mutex
}
```

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  Lock ml(&mutex);                    // from Item 14: acquire mutex and
                                      // ensure its later release
  delete bgImage;
  ++imageChanges;
  bgImage = new Image(imgSrc);
}
```

```
class PrettyMenu {
   ...
   std::tr1::shared_ptr<Image> bgImage;
   ...
};
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
   Lock ml(&mutex);
   bgImage.reset(new Image(imgSrc));       // replace bgImage's internal
                                           // pointer with the result of the
                                           // "new Image" expression
   ++imageChanges;
}
```

```
struct PMImpl {                                 // PMImpl = "PrettyMenu
   std::tr1::shared_ptr<Image> bgImage;         // Impl."; see below for
   int imageChanges;                            // why it's a struct
};

class PrettyMenu {
   ...

private:
   Mutex mutex;
   std::tr1::shared_ptr<PMImpl> pImpl;
};

void PrettyMenu::changeBackground(std::istream& imgSrc)
{
   using std::swap;                              // see Item 25

   Lock ml(&mutex);                              // acquire the mutex

   std::tr1::shared_ptr<PMImpl>                  // copy obj. data
      pNew(new PMImpl(*pImpl));

   pNew->bgImage.reset(new Image(imgSrc));       // modify the copy
   ++pNew->imageChanges;

   swap(pImpl, pNew);                            // swap the new
                                                 // data into place

}                                                // release the mutex
```

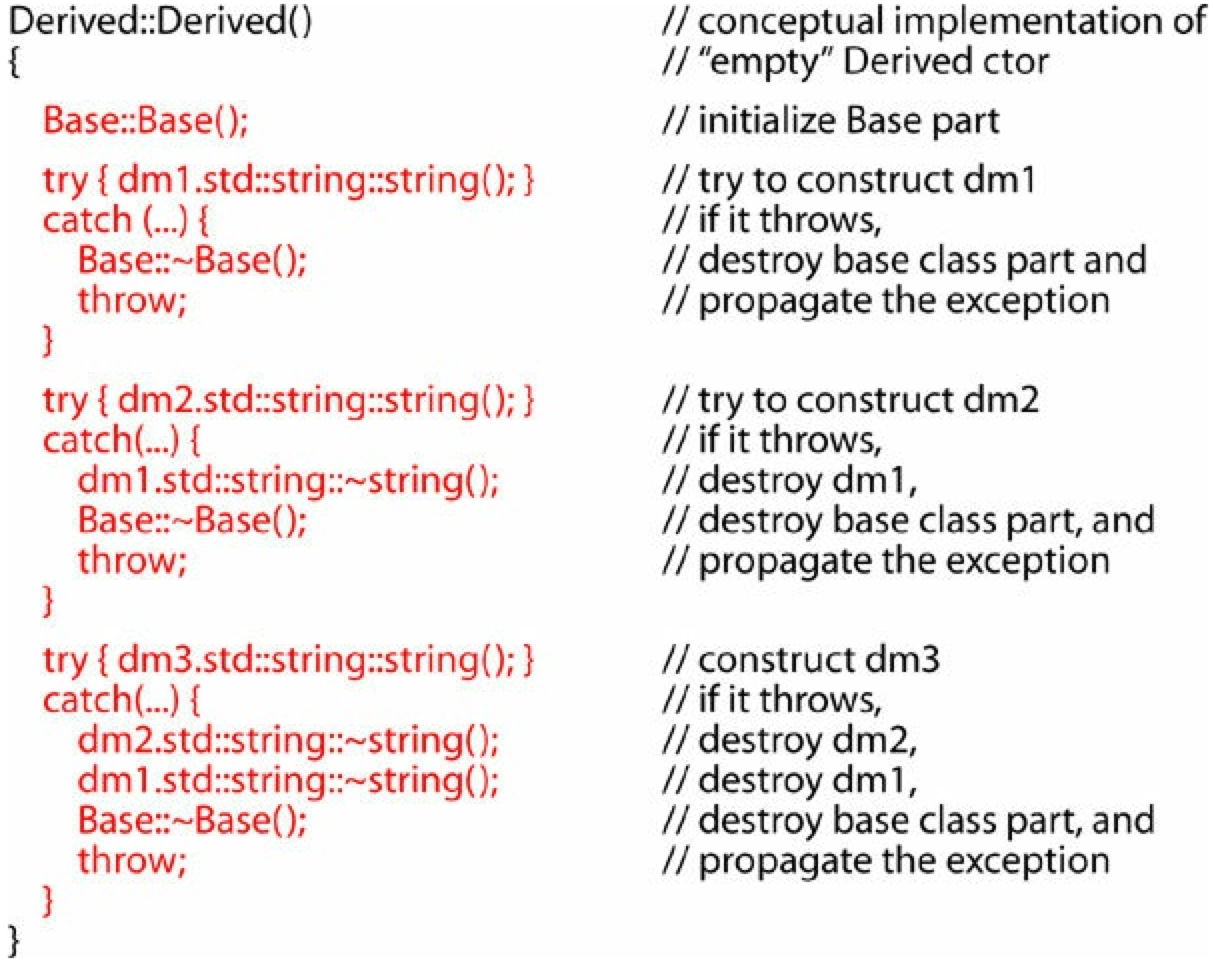
```
void someFunc()
{
   ...                                           // make copy of local state
   f1();
   f2();
   ...                                           // swap modified state into place
}
```

```
class Person {
public:
  ...
  int age() const { return theAge; }        // an implicit inline request: age is
  ...                                       // defined in a class definition

private:
  int theAge;
};
```

```
template<typename T>                                 // an explicit inline
inline const T& std::max(const T& a, const T& b)     // request: std::max is
{ return a < b ? b : a; }                            // preceded by “inline”
```

```
inline void f() {...}      // assume compilers are willing to inline calls to f
void (*pf)() = f;          // pf points to f
...
f();                       // this call will be inlined, because it's a "normal" call
pf();                      // this call probably won't be, because it's through
                           // a function pointer
```

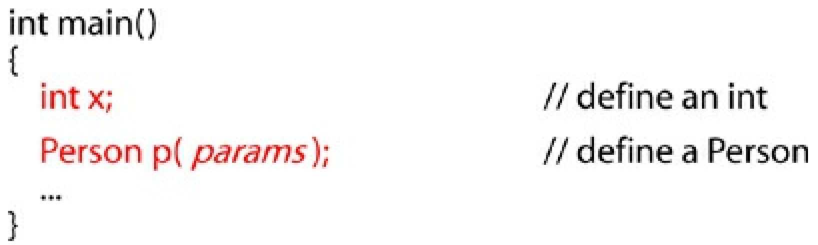
```
class Base {
public:
   ...

private:
   std::string bm1, bm2;                    // base members 1 and 2
};

class Derived: public Base {
public:
   Derived() {}                             // Derived's ctor is empty — or is it?
   ...

private:
   std::string dm1, dm2, dm3;               // derived members 1–3
};
```



```
Derived::Derived()                        // conceptual implementation of
{                                         // "empty" Derived ctor
  Base::Base();                           // initialize Base part

  try { dm1.std::string::string(); }      // try to construct dm1
  catch (...) {                           // if it throws,
    Base::~Base();                        // destroy base class part and
    throw;                                // propagate the exception
  }

  try { dm2.std::string::string(); }      // try to construct dm2
  catch(...) {                            // if it throws,
    dm1.std::string::~string();           // destroy dm1,
    Base::~Base();                        // destroy base class part, and
    throw;                                // propagate the exception
  }

  try { dm3.std::string::string(); }      // construct dm3
  catch(...) {                            // if it throws,
    dm2.std::string::~string();           // destroy dm2,
    dm1.std::string::~string();           // destroy dm1,
    Base::~Base();                        // destroy base class part, and
    throw;                                // propagate the exception
  }
}
```

```
class Person {
public:
   Person(const std::string& name, const Date& birthday,
            const Address& addr);
   std::string name() const;
   std::string birthDate() const;
   std::string address() const;
   ...
private:
   std::string theName;                    // implementation detail
   Date theBirthDate;                      // implementation detail
   Address theAddress;                     // implementation detail
};
```

```
namespace std {
  class string;                          // forward declaration (an incorrect
}                                        // one — see below)

class Date;                              // forward declaration
class Address;                           // forward declaration

class Person {
public:
  Person(const std::string& name, const Date& birthday,
         const Address& addr);
  std::string name() const;
  std::string birthDate() const;
  std::string address() const;
  ...
};
```



```
int main()
{
   int x;                                // define an int
   Person p( params );                   // define a Person
   ...
}
```

```
int main()
{
   int x;                              // define an int
   Person *p;                          // define a pointer to a Person
   ...
}
```

```
#include <string>                      // standard library components
                                       // shouldn't be forward-declared

#include <memory>                      // for tr1::shared_ptr; see below

class PersonImpl;                      // forward decl of Person impl. class

class Date;                            // forward decls of classes used in
class Address;                         // Person interface

class Person {
public:
  Person(const std::string& name, const Date& birthday,
         const Address& addr);
  std::string name() const;
  std::string birthDate() const;
  std::string address() const;
  ...

private:                                          // ptr to implementation;
  std::tr1::shared_ptr<PersonImpl> pImpl;         // see Item 13 for info on
};                                                // std::tr1::shared_ptr
```

```
class Date;                           // class declaration

Date today();                         // fine — no definition
void clearAppointments(Date d);       // of Date is needed
```

```
#include "datefwd.h"                  // header file declaring (but not
                                      // defining) class Date

Date today();                         // as before
void clearAppointments(Date d);
```

```
#include "Person.h"             // we're implementing the Person class,
                                // so we must #include its class definition
#include "PersonImpl.h"         // we must also #include PersonImpl's class
                                // definition, otherwise we couldn't call
                                // its member functions; note that
                                // PersonImpl has exactly the same public
                                // member functions as Person — their
                                // interfaces are identical
Person::Person(const std::string& name, const Date& birthday,
               const Address& addr)
: pImpl(new PersonImpl(name, birthday, addr))
{}
std::string Person::name() const
{
  return pImpl->name();
}
```

```
class Person {
public:
   virtual ~Person();

   virtual std::string name() const = 0;
   virtual std::string birthDate() const = 0;
   virtual std::string address() const = 0;
   ...
};
```

```
class Person {
public:
  ...
  static std::tr1::shared_ptr<Person>    // return a tr1::shared_ptr to a new
    create(const std::string& name,      // Person initialized with the
           const Date& birthday,         // given params; see Item 18 for
           const Address& addr);         // why a tr1::shared_ptr is returned
  ...
};
```

```
std::string name;
Date dateOfBirth;
Address address;
...
// create an object supporting the Person interface
std::tr1::shared_ptr<Person> pp(Person::create(name, dateOfBirth, address));

...
std::cout << pp->name()                  // use the object via the
          << " was born on "             // Person interface
          << pp->birthDate()
          << " and now lives at "
          << pp->address();
...                                      // the object is automatically
                                         // deleted when pp goes out of
                                         // scope — see Item 13
```

```
class RealPerson: public Person {
public:
   RealPerson(const std::string& name, const Date& birthday,
                     const Address& addr)
   :  theName(name), theBirthDate(birthday), theAddress(addr)
   {}

   virtual ~RealPerson() {}

   std::string name() const;             // implementations of these
   std::string birthDate() const;        // functions are not shown, but
   std::string address() const;          // they are easy to imagine

private:
   std::string theName;
   Date theBirthDate;
   Address theAddress;
};
```

```
std::tr1::shared_ptr<Person> Person::create(const std::string& name,
                                            const Date& birthday,
                                            const Address& addr)
{
  return std::tr1::shared_ptr<Person>(new RealPerson( name, birthday,
                                                      addr));
}
```

```
void eat(const Person& p);            // anyone can eat

void study(const Student& s);         // only students study

Person p;                             // p is a Person
Student s;                            // s is a Student

eat(p);                               // fine, p is a Person

eat(s);                               // fine, s is a Student,
                                      // and a Student is-a Person

study(s);                             // fine

study(p);                             // error! p isn’t a Student
```

```
class Bird {
public:
   virtual void fly();                          // birds can fly

   ...

};

class Penguin: public Bird {                     // penguins are birds

   ...

};
```

```
class Bird {
   ...                                     // no fly function is declared
};
class FlyingBird: public Bird {
public:
   virtual void fly();
   ...
};
class Penguin: public Bird {
   ...                                     // no fly function is declared
};
```

```
void error(const std::string& msg);          // defined elsewhere

class Penguin: public Bird {
public:
  virtual void fly() { error("Attempt to make a penguin fly!"); }

  ...

};
```

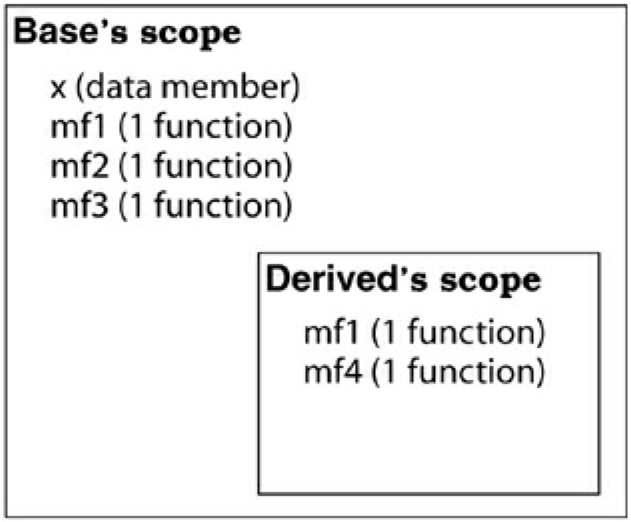
```
class Bird {
   ...                                    // no fly function is declared
};
class Penguin: public Bird {
   ...                                    // no fly function is declared
};
```

```
class Rectangle {
public:
   virtual void setHeight(int newHeight);
   virtual void setWidth(int newWidth);

   virtual int height() const;                  // return current values
   virtual int width() const;

   ...

};

void makeBigger(Rectangle& r)                   // function to increase r's area
{
   int oldHeight = r.height();

   r.setWidth(r.width() + 10);                  // add 10 to r's width

   assert(r.height() == oldHeight);             // assert that r's
}                                               // height is unchanged
```

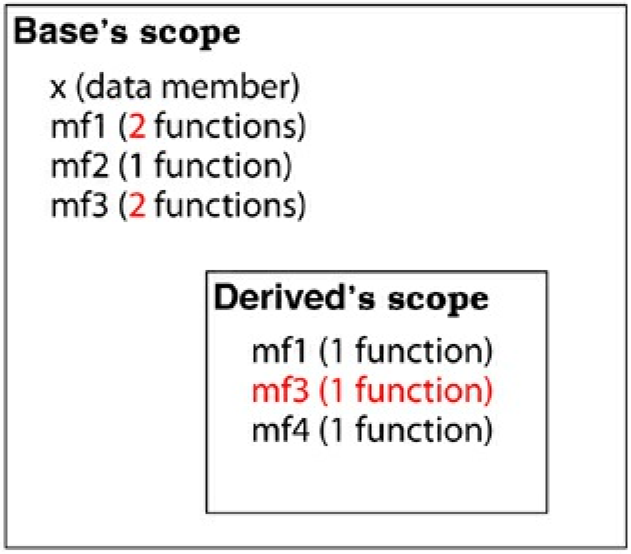
```
class Square: public Rectangle { ... };
Square s;
...
assert(s.width() == s.height());        // this must be true for all squares
makeBigger(s);                          // by inheritance, s is-a Rectangle,
                                        // so we can increase its area
assert(s.width() == s.height());        // this must still be true
                                        // for all squares
```

```
int x;                                 // global variable

void someFunc()
{
   double x;                           // local variable

   std::cin >> x;                      // read a new value for local x
}
```

```
class Base {
private:
   int x;

public:
   virtual void mf1() = 0;
   virtual void mf2();
   void mf3();
   ...
};

class Derived: public Base {
public:
   virtual void mf1();
   void mf4();
   ...
};
```

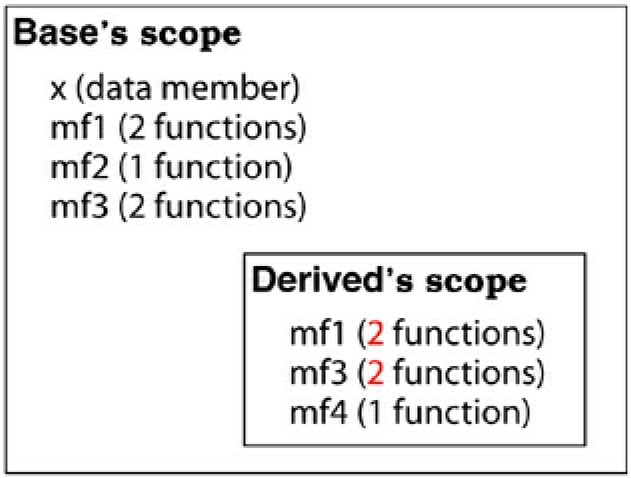

Base's scope
x (data member)
mf1 (1 function)
mf2 (1 function)
mf3 (1 function)
Derived's scope
mf1 (1 function)
mf4 (1 function)

```
class Base {
private:
   int x;

public:
   virtual void mf1() = 0;
   virtual void mf1(int);

   virtual void mf2();

   void mf3();
   void mf3(double);
   ...
};

class Derived: public Base {
public:
   virtual void mf1();
   void mf3();
   void mf4();
   ...
};
```


Base's scope
x (data member)
mf1 (2 functions)
mf2 (1 function)
mf3 (2 functions)
Derived's scope
mf1 (1 function)
mf3 (1 function)
mf4 (1 function)

```
Derived d;
int x;

...

d.mf1();                    // fine, calls Derived::mf1
d.mf1(x);                   // error! Derived::mf1 hides Base::mf1
d.mf2();                    // fine, calls Base::mf2

d.mf3();                    // fine, calls Derived::mf3
d.mf3(x);                   // error! Derived::mf3 hides Base::mf3
```

```
class Base {
private:
   int x;

public:
   virtual void mf1() = 0;
   virtual void mf1(int);

   virtual void mf2();

   void mf3();
   void mf3(double);
   ...
};

class Derived: public Base {
public:
   using Base::mf1;         // make all things in Base named mf1 and mf3
   using Base::mf3;         // visible (and public) in Derived's scope

   virtual void mf1();
   void mf3();
   void mf4();
   ...
};
```



Base's scope
x (data member)
mf1 (2 functions)
mf2 (1 function)
mf3 (2 functions)
Derived's scope
mf1 (2 functions)
mf3 (2 functions)
mf4 (1 function)

```
class Base {
public:
  virtual void mf1() = 0;
  virtual void mf1(int);

  ...                                   // as before
};

class Derived: private Base {
public:
  virtual void mf1()                    // forwarding function; implicitly
  { Base::mf1(); }                      // inline — see Item 30. (For info
  ...                                   // on calling a pure virtual
};                                      // function, see Item 34.)

...

Derived d;
int x;

d.mf1();                                // fine, calls Derived::mf1
d.mf1(x);                               // error! Base::mf1() is hidden
```

```
class Shape {
public:
   virtual void draw() const = 0;
   virtual void error(const std::string& msg);
   int objectID() const;
   ...
};
class Rectangle: public Shape { ... };
class Ellipse: public Shape { ... };
```

```
Shape *ps = new Shape;              // error! Shape is abstract

Shape *ps1 = new Rectangle;         // fine
ps1->draw();                        // calls Rectangle::draw

Shape *ps2 = new Ellipse;           // fine
ps2->draw();                        // calls Ellipse::draw

ps1->Shape::draw();                 // calls Shape::draw

ps2->Shape::draw();                 // calls Shape::draw
```

```
class Shape {
public:
   virtual void error(const std::string& msg);
   ...
};
```

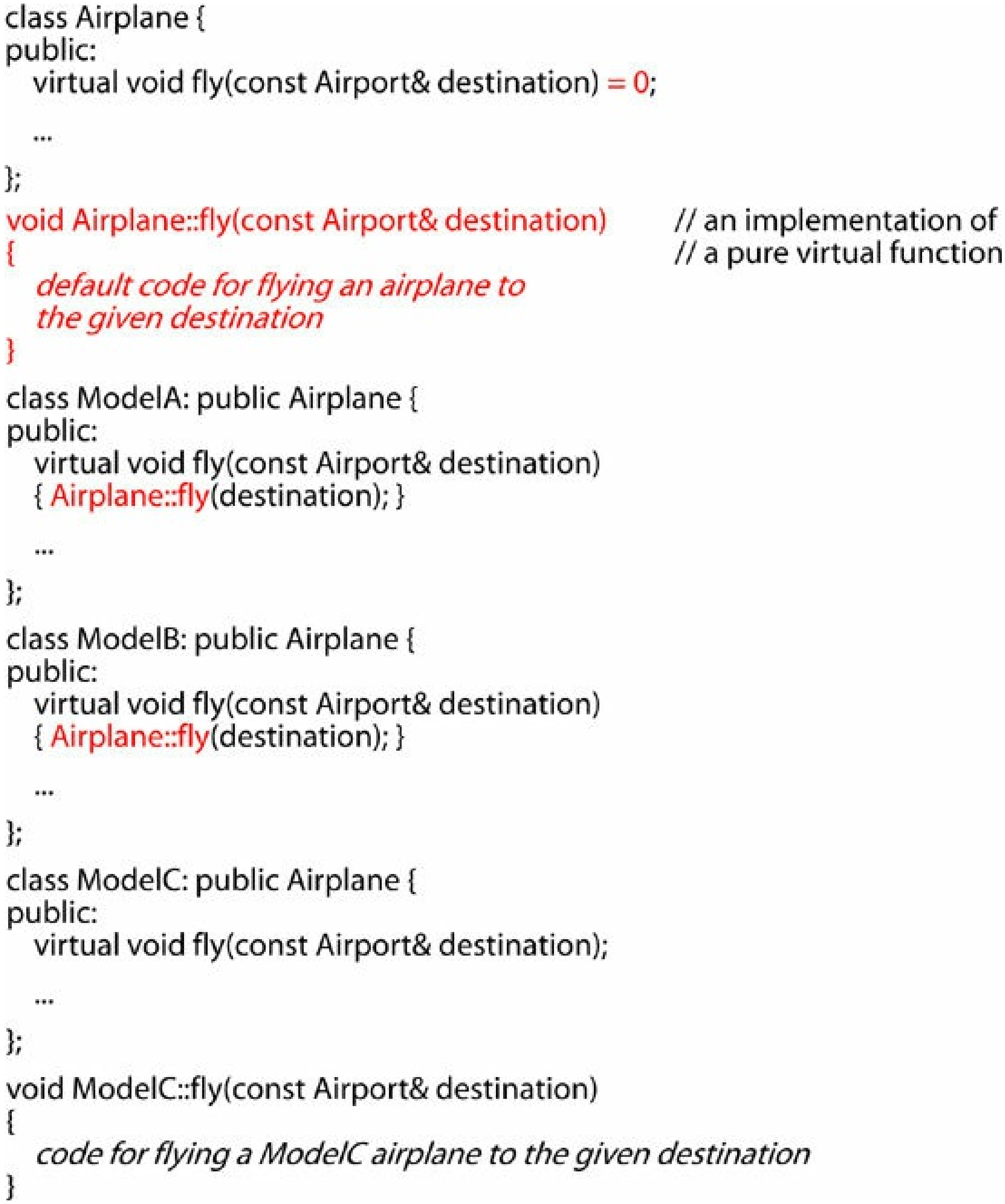
```
class Airport { ... };                          // represents airports

class Airplane {
public:
  virtual void fly(const Airport& destination);

  ...

};

void Airplane::fly(const Airport& destination)
{
  default code for flying an airplane to the given destination
}

class ModelA: public Airplane { ... };

class ModelB: public Airplane { ... };
class ModelC: public Airplane {
   ...                                          // no fly function is declared
};
```

```
Airport PDX(...);                       // PDX is the airport near my home
Airplane *pa = new ModelC;
...
pa->fly(PDX);                           // calls Airplane::fly!
```

```
class Airplane {
public:
   virtual void fly(const Airport& destination) = 0;

   ...

protected:
   void defaultFly(const Airport& destination);
};

void Airplane::defaultFly(const Airport& destination)
{
   default code for flying an airplane to the given destination
}
```

```
class ModelA: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { defaultFly(destination); }

   ...

};

class ModelB: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { defaultFly(destination); }

   ...

};
```

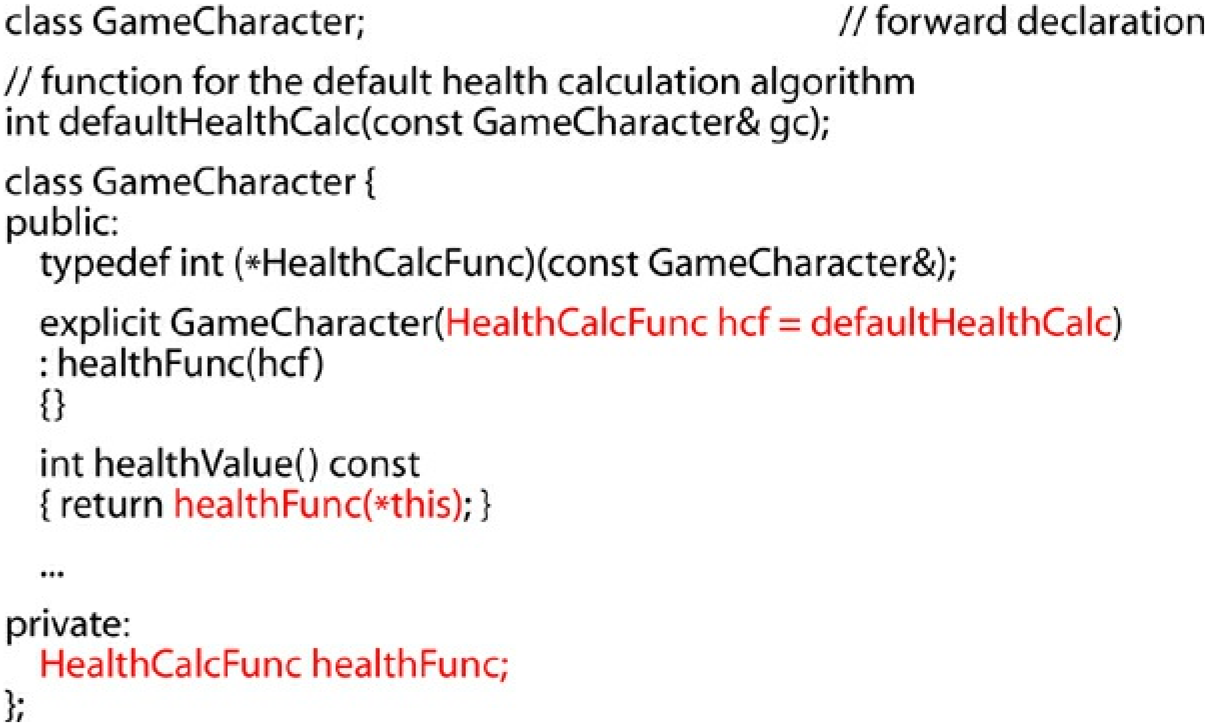
```
class ModelC: public Airplane {
public:
   virtual void fly(const Airport& destination);

   ...

};

void ModelC::fly(const Airport& destination)
{
   code for flying a ModelC airplane to the given destination
}
```



```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;

  ...

};
void Airplane::fly(const Airport& destination)       // an implementation of
{                                                     // a pure virtual function
  default code for flying an airplane to
  the given destination
}
class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }

  ...

};
class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }

  ...

};
class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);

  ...

};
void ModelC::fly(const Airport& destination)
{
  code for flying a ModelC airplane to the given destination
}
```

```
class GameCharacter {
public:
  virtual int healthValue() const;        // return character's health rating;
  ...                                     // derived classes may redefine this
};
```

```
class GameCharacter {
public:
  int healthValue() const               // derived classes do not redefine
  {                                     // this — see Item 36

    ...                                 // do "before" stuff — see below

    int retVal = doHealthValue();       // do the real work

    ...                                 // do "after" stuff — see below

    return retVal;
  }
  ...

private:
  virtual int doHealthValue() const     // derived classes may redefine this
  {
    ...                                 // default algorithm for calculating
  }                                     // character's health
};
```



```
class GameCharacter;                              // forward declaration

// function for the default health calculation algorithm
int defaultHealthCalc(const GameCharacter& gc);

class GameCharacter {
public:
  typedef int (*HealthCalcFunc)(const GameCharacter&);

  explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc)
  : healthFunc(hcf )
  {}

  int healthValue() const
  { return healthFunc(*this); }

  ...

private:
  HealthCalcFunc healthFunc;
};
```

```
class EvilBadGuy: public GameCharacter {
public:
  explicit EvilBadGuy(HealthCalcFunc hcf = defaultHealthCalc)
  : GameCharacter(hcf)
  { ... }

  ...

};

int loseHealthQuickly(const GameCharacter&);      // health calculation
int loseHealthSlowly(const GameCharacter&);       // funcs with different
                                                  // behavior

EvilBadGuy ebg1(loseHealthQuickly);               // same-type charac-
EvilBadGuy ebg2(loseHealthSlowly);                // ters with different
                                                  // health-related
                                                  // behavior
```

```
class GameCharacter;                                      // as before
int defaultHealthCalc(const GameCharacter& gc);           // as before

class GameCharacter {
public:
  // HealthCalcFunc is any callable entity that can be called with
  // anything compatible with a GameCharacter and that returns anything
  // compatible with an int; see below for details
  typedef std::tr1::function<int (const GameCharacter&)> HealthCalcFunc;
  explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc)
  : healthFunc(hcf )
  {}

  int healthValue() const
  { return healthFunc(*this); }

  ...

private:
  HealthCalcFunc healthFunc;
};
```

```
short calcHealth(const GameCharacter&);             // health calculation
                                                    // function; note
                                                    // non-int return type

struct HealthCalculator {                           // class for health
  int operator()(const GameCharacter&) const        // calculation function
  { ... }                                           // objects
};

class GameLevel {
public:
  float health(const GameCharacter&) const;         // health calculation
  ...                                               // mem function; note
};                                                  // non-int return type


class EvilBadGuy: public GameCharacter {            // as before
  ...
};

class EyeCandyCharacter: public GameCharacter {     // another character
  ...                                               // type; assume same
};                                                  // constructor as
                                                    // EvilBadGuy


EvilBadGuy ebg1(calcHealth);                        // character using a
                                                    // health calculation
                                                    // function

EyeCandyCharacter ecc1(HealthCalculator());         // character using a
                                                    // health calculation
                                                    // function object

GameLevel currentLevel;
...
EvilBadGuy ebg2(                                    // character using a
  std::tr1::bind(&GameLevel::health,                // health calculation
                 currentLevel,                      // member function;
                 _1)                                // see below for details
);
```

```
class GameCharacter;                              // forward declaration

class HealthCalcFunc {
public:
   ...
   virtual int calc(const GameCharacter& gc) const
   { ... }
   ...
};

HealthCalcFunc defaultHealthCalc;

class GameCharacter {
public:
   explicit GameCharacter(HealthCalcFunc *phcf = &defaultHealthCalc)
   : pHealthCalc(phcf)
   {}

   int healthValue() const
   { return pHealthCalc->calc(*this); }

   ...

private:
   HealthCalcFunc *pHealthCalc;
};
```

```
B *pB = &x;                      // get pointer to x
pB->mf();                        // call mf through pointer
```

```
D *pD = &x;                             // get pointer to x
pD->mf();                               // call mf through pointer
```

```
class D: public B {
public:
  void mf();                            // hides B::mf; see Item 33

  ...

};

pB->mf();                               // calls B::mf

pD->mf();                               // calls D::mf
```

```
// a class for geometric shapes
class Shape {
public:
   enum ShapeColor { Red, Green, Blue };

   // all shapes must offer a function to draw themselves
   virtual void draw(ShapeColor color = Red) const = 0;
   ...
};

class Rectangle: public Shape {
public:
   // notice the different default parameter value — bad!
   virtual void draw(ShapeColor color = Green) const;
   ...
};

class Circle: public Shape {
public:
   virtual void draw(ShapeColor color) const;
   ...
};
```

```
Shape *ps;                        // static type = Shape*
Shape *pc = new Circle;           // static type = Shape*
Shape *pr = new Rectangle;        // static type = Shape*
```

```
ps = pc;                        // ps's dynamic type is
                                // now Circle*
ps = pr;                        // ps's dynamic type is
                                // now Rectangle*
```

```
pc->draw(Shape::Red);                    // calls Circle::draw(Shape::Red)
pr->draw(Shape::Red);                    // calls Rectangle::draw(Shape::Red)
```

```
class Shape {
public:
   enum ShapeColor { Red, Green, Blue };
   virtual void draw(ShapeColor color = Red) const = 0;
   ...
};
class Rectangle: public Shape {
public:
   virtual void draw(ShapeColor color = Red) const;
   ...
};
```

```
class Shape {
public:
   enum ShapeColor { Red, Green, Blue };

   void draw(ShapeColor color = Red) const              // now non-virtual
   {
      doDraw(color);                                     // calls a virtual
   }

   ...

private:
   virtual void doDraw(ShapeColor color) const = 0;      // the actual work is
};                                                       // done in this func

class Rectangle: public Shape {
public:
   ...

private:
   virtual void doDraw(ShapeColor color) const;          // note lack of a
   ...                                                   // default param val.
};
```

```
class Address { ... };                  // where someone lives

class PhoneNumber { ... };

class Person {
public:
  ...

private:
  std::string name;                     // composed object
  Address address;                      // ditto
  PhoneNumber voiceNumber;              // ditto
  PhoneNumber faxNumber;                // ditto
};
```

```
template<class T>                          // the right way to use list for Set
class Set {
public:
  bool member(const T& item) const;

  void insert(const T& item);
  void remove(const T& item);

  std::size_t size() const;

private:
  std::list<T> rep;                        // representation for Set data
};
```

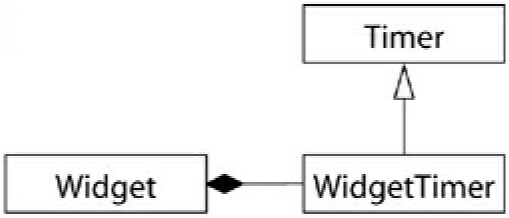
```
template<typename T>
bool Set<T>::member(const T& item) const
{
  return std::find(rep.begin(), rep.end(), item) != rep.end();
}

template<typename T>
void Set<T>::insert(const T& item)
{
  if (!member(item)) rep.push_back(item);
}

template<typename T>
void Set<T>::remove(const T& item)
{
  typename std::list<T>::iterator it =               // see Item 42 for info on
    std::find(rep.begin(), rep.end(), item);         // "typename" here

  if (it != rep.end()) rep.erase(it);
}

template<typename T>
std::size_t Set<T>::size() const
{
  return rep.size();
}
```

```
class Person { ... };
class Student: private Person { ... };      // inheritance is now private
void eat(const Person& p);                  // anyone can eat
void study(const Student& s);               // only students study
Person p;                                   // p is a Person
Student s;                                  // s is a Student
eat(p);                                     // fine, p is a Person
eat(s);                                     // error! a Student isn't a Person
```

```
class Timer {
public:
   explicit Timer(int tickFrequency);
    virtual void onTick() const;               // automatically called for each tick
   ...
};
```

```
class Widget: private Timer {
private:
   virtual void onTick() const;                 // look at Widget usage data, etc.
   ...
};
```

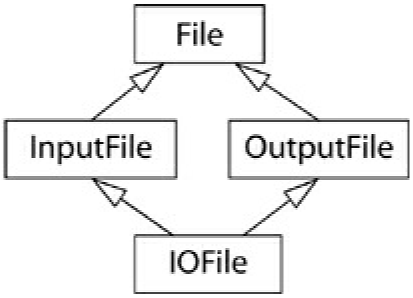
```
class Widget {
private:
   class WidgetTimer: public Timer {
   public:
      virtual void onTick() const;
      ...
   };

   WidgetTimer timer;
   ...
};
```


Timer
Widget
WidgetTimer

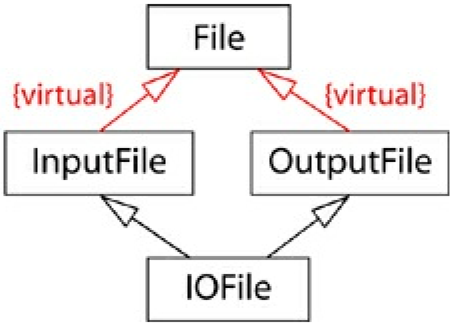
```
class Empty {};                    // has no data, so objects should
                                   // use no memory

class HoldsAnInt {                 // should need only space for an int
private:
  int x;
  Empty e;                         // should require no memory
};
```

```
class BorrowableItem {                    // something a library lets you borrow
public:
  void checkOut();                        // check the item out from the library
  ...
};

class ElectronicGadget {
private:
  bool checkOut() const;                  // perform self-test, return whether
  ...                                     // test succeeds
};

class MP3Player:                          // note MI here
  public BorrowableItem,                  // (some libraries loan MP3 players)
  public ElectronicGadget
{ ... };                                  // class definition is unimportant

MP3Player mp;

mp.checkOut();                            // ambiguous! which checkOut?
```

```
class File { ... };
class InputFile: public File { ... };
class OutputFile: public File { ... };
class IOFile: public InputFile,
              public OutputFile
{ ... };
```


File
InputFile
OutputFile
IOFile

```
class File { ... };
class InputFile: virtual public File { ... };
class OutputFile: virtual public File { ... };
class IOFile: public InputFile,
              public OutputFile
{ ... };
```

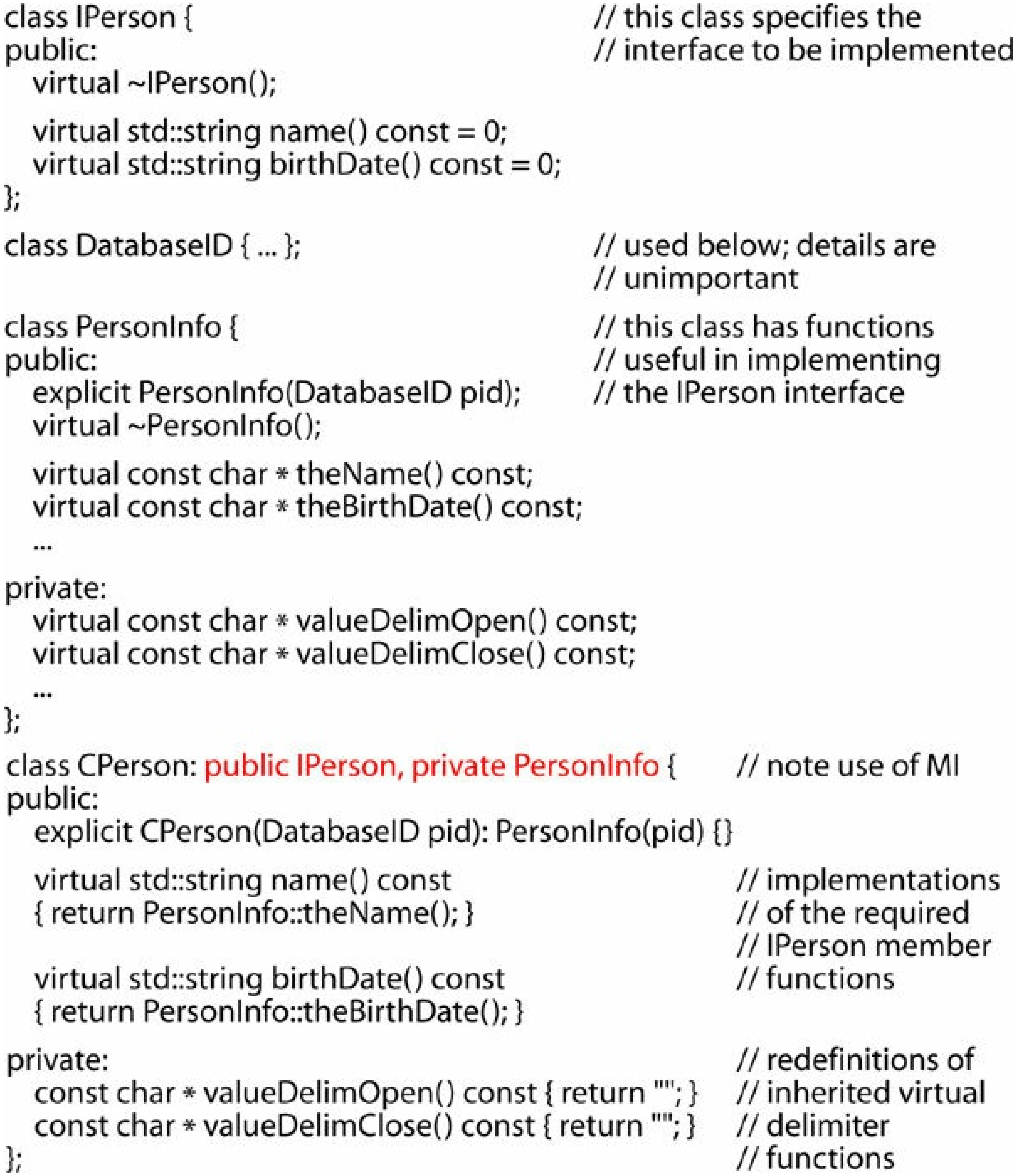

File
{virtual}
{virtual}
InputFile
OutputFile
IOFile

```
class IPerson {
public:
   virtual ~IPerson();

   virtual std::string name() const = 0;
   virtual std::string birthDate() const = 0;
};
```

```
// factory function to create a Person object from a unique database ID;
// see Item 18 for why the return type isn't a raw pointer
std::tr1::shared_ptr<IPerson> makePerson(DatabaseID personIdentifier);

// function to get a database ID from the user
DatabaseID askUserForDatabaseID();


DatabaseID id(askUserForDatabaseID());

std::tr1::shared_ptr<IPerson> pp(makePerson(id));     // create an object
                                                      // supporting the
                                                      // IPerson interface

...                                                   // manipulate *pp via
                                                      // IPerson's member
                                                      // functions
```

```
class PersonInfo {
public:
  explicit PersonInfo(DatabaseID pid);
  virtual ~PersonInfo();

  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  ...

private:
  virtual const char * valueDelimOpen() const;        // see
  virtual const char * valueDelimClose() const;       // below
  ...
};
```

```
const char * PersonInfo::valueDelimOpen() const
{
  return "[";                              // default opening delimiter
}

const char * PersonInfo::valueDelimClose() const
{
  return "]";                              // default closing delimiter
}

const char * PersonInfo::theName() const
{
  // reserve buffer for return value; because this is
  // static, it's automatically initialized to all zeros
  static char value[Max_Formatted_Field_Value_Length];

  // write opening delimiter
  std::strcpy(value, valueDelimOpen());

  append to the string in value this object's name field (being careful
  to avoid buffer overruns!)

  // write closing delimiter
  std::strcat(value, valueDelimClose());

  return value;
}
```

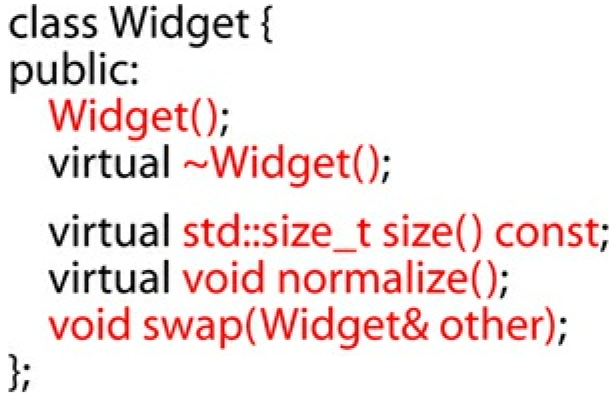


```
class IPerson {                                   // this class specifies the
public:                                           // interface to be implemented
  virtual ~IPerson();

  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
};

class DatabaseID { ... };                         // used below; details are
                                                  // unimportant

class PersonInfo {                                // this class has functions
public:                                           // useful in implementing
  explicit PersonInfo(DatabaseID pid);            // the IPerson interface
  virtual ~PersonInfo();

  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  ...

private:
  virtual const char * valueDelimOpen() const;
  virtual const char * valueDelimClose() const;
  ...
};

class CPerson: public IPerson, private PersonInfo {       // note use of MI
public:
  explicit CPerson(DatabaseID pid): PersonInfo(pid) {}

  virtual std::string name() const                         // implementations
  { return PersonInfo::theName(); }                        // of the required
                                                           // IPerson member
  virtual std::string birthDate() const                    // functions
  { return PersonInfo::theBirthDate(); }

private:                                                   // redefinitions of
  const char * valueDelimOpen() const { return ""; }       // inherited virtual
  const char * valueDelimClose() const { return ""; }      // delimiter
};                                                         // functions
```

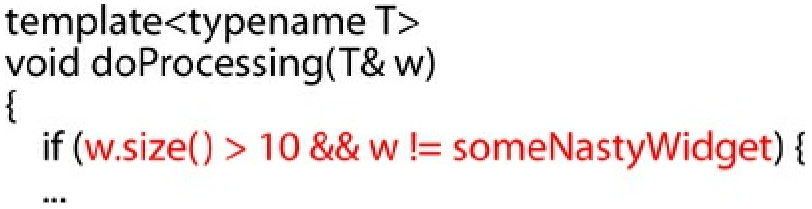
```
class Widget {
public:
   Widget();
   virtual ~Widget();
   virtual std::size_t size() const;
   virtual void normalize();
   void swap(Widget& other);                          // see Item 25

   ...
};
```

```
void doProcessing(Widget& w)
{
   if (w.size() > 10 && w != someNastyWidget) {
      Widget temp(w);
      temp.normalize();
      temp.swap(w);
   }
}
```

```
template<typename T>
void doProcessing(T& w)
{
   if (w.size() > 10 && w != someNastyWidget) {
      T temp(w);
      temp.normalize();
      temp.swap(w);
   }
}
```



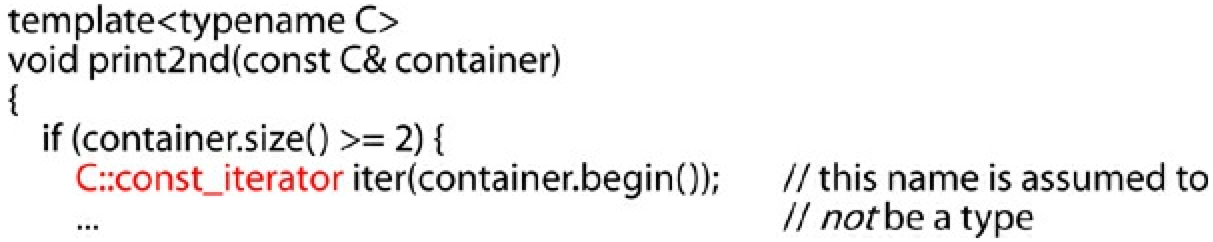
```
class Widget {
public:
   Widget();
   virtual ~Widget();

   virtual std::size_t size() const;
   virtual void normalize();
   void swap(Widget& other);
};
```



```
template<typename T>
void doProcessing(T& w)
{
   if (w.size() > 10 && w != someNastyWidget) {
   ...
```

```
template<class T> class Widget;                    // uses "class"
template<typename T> class Widget;                 // uses "typename"
```

```
template<typename C>                            // print 2nd element in
void print2nd(const C& container)               // container;
{                                               // this is not valid C++!
   if (container.size() >= 2) {
      C::const_iterator iter(container.begin()); // get iterator to 1st element
      ++iter;                                   // move iter to 2nd element
      int value = *iter;                        // copy that element to an int
      std::cout << value;                       // print the int
   }
}
```



```
template<typename C>
void print2nd(const C& container)
{
   if (container.size() >= 2) {
      C::const_iterator iter(container.begin());       // this name is assumed to
      ...                                              // not be a type
```

```
template<typename C>                          // this is valid C++
void print2nd(const C& container)
{
  if (container.size() >= 2) {
    typename C::const_iterator iter(container.begin());

    ...

  }
}
```

```
template<typename C>                          // typename allowed (as is "class")
void f(const C& container,                    // typename not allowed
       typename C::iterator iter);            // typename required
```

```
template<typename T>
class Derived: public Base<T>::Nested {   // base class list: typename not
public:                                   // allowed
   explicit Derived(int x)
   : Base<T>::Nested(x)                   // base class identifier in mem.
   {                                      // init. list: typename not allowed

      typename Base<T>::Nested temp;      // use of nested dependent type
      ...                                 // name not in a base class list or
   }                                      // as a base class identifier in a
   ...                                    // mem. init. list: typename
required
};
```

```
template<typename IterT>
void workWithIterator(IterT iter)
{
   typename std::iterator_traits<IterT>::value_type temp(*iter);
   ...
}
```

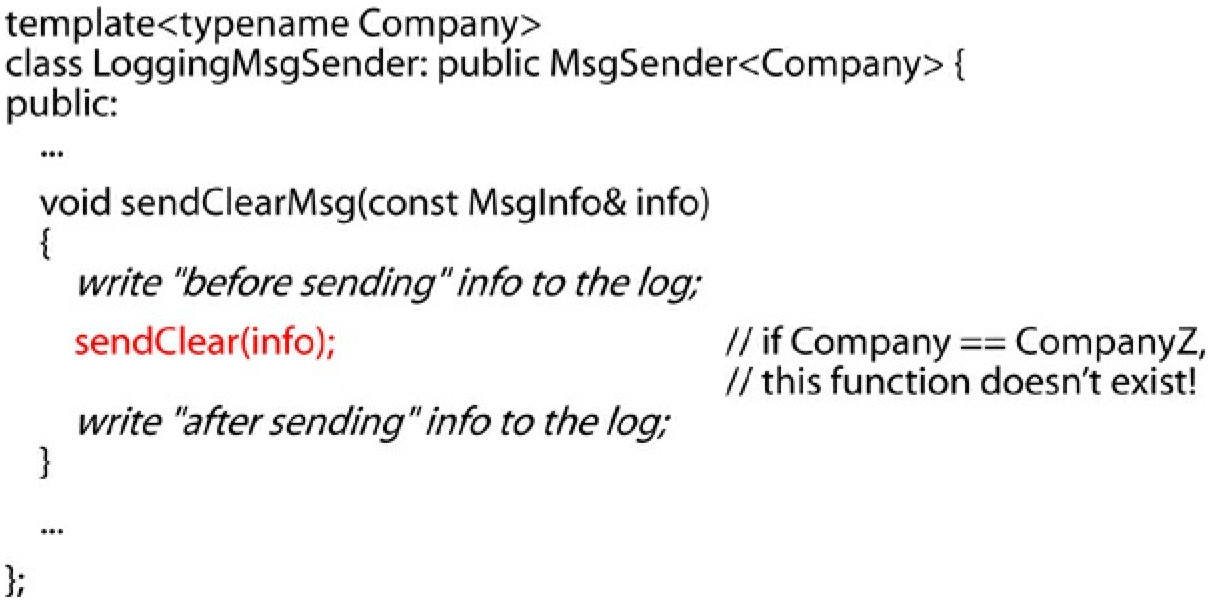
```
template<typename IterT>
void workWithIterator(IterT iter)
{
   typedef typename std::iterator_traits<IterT>::value_type value_type;
   value_type temp(*iter);
   ...
}
```

```
class CompanyA {
public:
  ...
  void sendCleartext(const std::string& msg);
  void sendEncrypted(const std::string& msg);
  ...
};

class CompanyB {
public:
  ...
  void sendCleartext(const std::string& msg);
  void sendEncrypted(const std::string& msg);
  ...
};

...                                        // classes for other companies

class MsgInfo { ... };                     // class for holding information
                                           // used to create a message

template<typename Company>
class MsgSender {
public:
  ...                                      // ctors, dtor, etc.
  void sendClear(const MsgInfo& info)
  {
    std::string msg;
    create msg from info;

    Company c;
    c.sendCleartext(msg);
  }

  void sendSecret(const MsgInfo& info)     // similar to sendClear, except
  { ... }                                  // calls c.sendEncrypted

};
```

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...                                            // ctors, dtor, etc.
  void sendClearMsg(const MsgInfo& info)
  {
    write "before sending" info to the log;

    sendClear(info);                             // call base class function;
                                                 // this code will not compile!
    write "after sending" info to the log;
  }
  ...
};
```

```
class CompanyZ {                                   // this class offers no
public:                                            // sendCleartext function
  ...
  void sendEncrypted(const std::string& msg);
  ...
};
```

```
template<>                                      // a total specialization of
class MsgSender<CompanyZ> {                     // MsgSender; the same as the
public:                                         // general template, except
  ...                                           // sendClear is omitted
  void sendSecret(const MsgInfo& info)
  { ... }

};
```



```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
   ...
   void sendClearMsg(const MsgInfo& info)
   {
      write "before sending" info to the log;
      sendClear(info);                              // if Company == CompanyZ,
                                                    // this function doesn't exist!
      write "after sending" info to the log;
   }

   ...

};
```

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
   ...
   void sendClearMsg(const MsgInfo& info)
   {
      write "before sending" info to the log;
      this->sendClear(info);                      // okay, assumes that
                                                  // sendClear will be inherited
      write "after sending" info to the log;
   }

   ...
};
```

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  using MsgSender<Company>::sendClear;   // tell compilers to assume
  ...                                    // that sendClear is in the
                                         // base class
  void sendClearMsg(const MsgInfo& info)
  {
    ...
    sendClear(info);                     // okay, assumes that
    ...                                  // sendClear will be inherited
  }

  ...

};
```

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
   ...
   void sendClearMsg(const MsgInfo& info)
   {
      ...
      MsgSender<Company>::sendClear(info);         // okay, assumes that
      ...                                           // sendClear will be
   }                                                // inherited

   ...

};
```

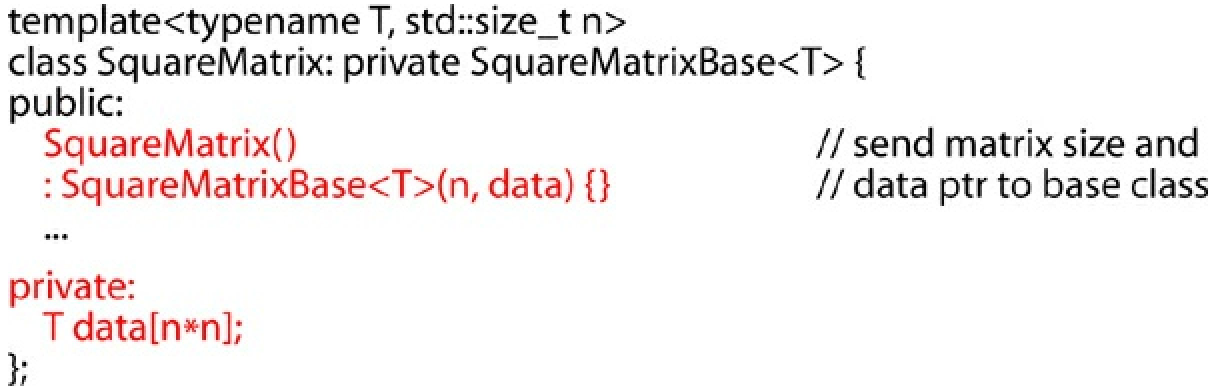
```
LoggingMsgSender<CompanyZ> zMsgSender;
MsgInfo msgData;
...                                     // put info in msgData
zMsgSender.sendClearMsg(msgData);       // error! won’t compile
```

```
template<typename T,                     // template for n x n matrices of
         std::size_t n>                  // objects of type T; see below for info
class SquareMatrix {                     // on the size_t parameter
public:
  ...
  void invert();                         // invert the matrix in place
};
```

```
SquareMatrix<double, 5> sm1;
...
sm1.invert();                          // call SquareMatrix<double, 5>::invert
SquareMatrix<double, 10> sm2;
...
sm2.invert();                          // call SquareMatrix<double, 10>::invert
```

```
template<typename T>                       // size-independent base class for
class SquareMatrixBase {                   // square matrices
protected:
  ...
  void invert(std::size_t matrixSize);     // invert matrix of the given size
  ...
};

template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
private:
  using SquareMatrixBase<T>::invert;       // make base class version of invert
                                           // visible in this class; see Items 33
                                           // and 43
public:
  ...
  void invert() { invert(n); }             // make inline call to base class
};                                         // version of invert
```

```
template<typename T>
class SquareMatrixBase {
protected:
   SquareMatrixBase(std::size_t n, T *pMem)       // store matrix size and a
   : size(n), pData(pMem) {}                      // ptr to matrix values

   void setDataPtr(T *ptr) { pData = ptr; }       // reassign pData
   ...

private:
   std::size_t size;                              // size of matrix

   T *pData;                                      // pointer to matrix values
};
```



```
template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
public:
   SquareMatrix()                                  // send matrix size and
   : SquareMatrixBase<T>(n, data) {}               // data ptr to base class
   ...
private:
   T data[n*n];
};
```

```
template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
public:
  SquareMatrix()                          // set base class data ptr to null,
  :  SquareMatrixBase<T>(n, 0),           // allocate memory for matrix
     pData(new T[n*n])                    // values, save a ptr to the
  { this->setDataPtr(pData.get()); }      // memory, and give a copy of it
  ...                                     // to the base class

private:
  boost::scoped_array<T> pData;           // see Item 13 for info on
};                                        // boost::scoped_array
```

```
class Top { ... };
class Middle: public Top { ... };
class Bottom: public Middle { ... };
Top *pt1 = new Middle;             // convert Middle* ⇒ Top*
Top *pt2 = new Bottom;             // convert Bottom* ⇒ Top*
const Top *pct2 = pt1;             // convert Top* ⇒ const Top*
```

```
template<typename T>
class SmartPtr {
public:                                   // smart pointers are typically
  explicit SmartPtr(T *realPtr);          // initialized by built-in pointers
  ...
};

SmartPtr<Top> pt1 =                       // convert SmartPtr<Middle> ⇒
  SmartPtr<Middle>(new Middle);           //    SmartPtr<Top>

SmartPtr<Top> pt2 =                       // convert SmartPtr<Bottom> ⇒
  SmartPtr<Bottom>(new Bottom);           //    SmartPtr<Top>

SmartPtr<const Top> pct2 = pt1;           // convert SmartPtr<Top> ⇒
                                          //    SmartPtr<const Top>
```

```
template<typename T>
class SmartPtr {
public:
   template<typename U>                        // member template
   SmartPtr(const SmartPtr<U>& other);         // for a ”generalized
   ...                                         // copy constructor”
};
```

```
template<typename T>
class SmartPtr {
public:
   template<typename U>
   SmartPtr(const SmartPtr<U>& other)              // initialize this held ptr
   : heldPtr(other.get()) { ... }                  // with other's held ptr

   T* get() const { return heldPtr; }
   ...

private:                                           // built-in pointer held
   T *heldPtr;                                     // by the SmartPtr
};
```

```
template<class T> class shared_ptr {
public:
  template<class Y>                                       // construct from
    explicit shared_ptr(Y * p);                           // any compatible
  template<class Y>                                       // built-in pointer,
    shared_ptr(shared_ptr<Y> const& r);                   // shared_ptr,
  template<class Y>                                       // weak_ptr, or
    explicit shared_ptr(weak_ptr<Y> const& r);            // auto_ptr
  template<class Y>
    explicit shared_ptr(auto_ptr<Y>& r);
  template<class Y>                                       // assign from
    shared_ptr& operator=(shared_ptr<Y> const& r);        // any compatible
  template<class Y>                                       // shared_ptr or
    shared_ptr& operator=(auto_ptr<Y>& r);                // auto_ptr
  ...
};
```

```
template<class T> class shared_ptr {
public:
  shared_ptr(shared_ptr const& r);                    // copy constructor

  template<class Y>                                   // generalized
    shared_ptr(shared_ptr<Y> const& r);               // copy constructor

  shared_ptr& operator=(shared_ptr const& r);         // copy assignment

  template<class Y>                                   // generalized
    shared_ptr& operator=(shared_ptr<Y> const& r);    // copy assignment
  ...
};
```

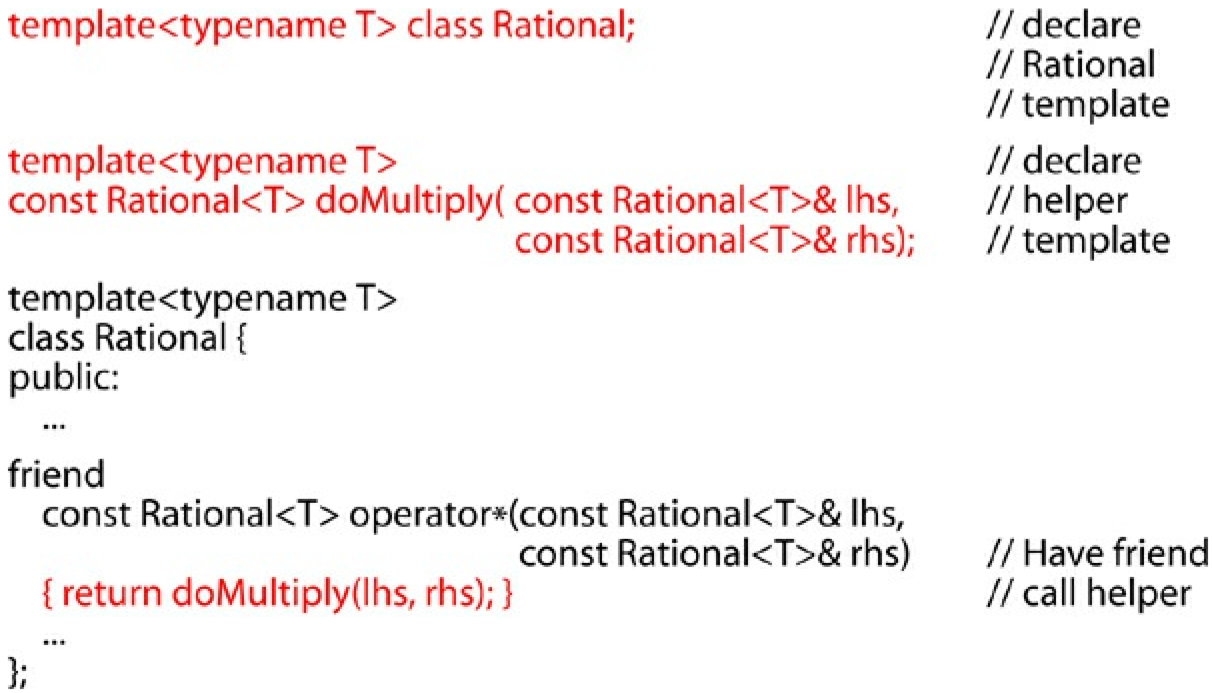
```
template<typename T>
class Rational {
public:
  Rational(const T& numerator = 0,        // see Item 20 for why params
           const T& denominator = 1);     // are now passed by reference

  const T numerator() const;              // see Item 28 for why return
  const T denominator() const;            // values are still passed by value,
  ...                                     // Item 3 for why they're const

};

template<typename T>
const Rational<T> operator*(const Rational<T>& lhs,
                            const Rational<T>& rhs)
{ ... }
```

```
Rational<int> oneHalf(1, 2);            // this example is from Item 24,
                                        // except Rational is now a template

Rational<int> result = oneHalf * 2;     // error! won't compile
```

```
template<typename T>
class Rational {
public:
  ...
friend                                                    // declare operator*
  const Rational operator*(const Rational& lhs,           // function (see
                           const Rational& rhs);          // below for details)
};

template<typename T>                                      // define operator*
const Rational<T> operator*(const Rational<T>& lhs,       // functions
                            const Rational<T>& rhs)
{ ... }
```

```
template<typename T>
class Rational {
public:
   ...
friend
   const Rational<T> operator*(const Rational<T>& lhs,
                               const Rational<T>& rhs);
   ...
};
```

```
template<typename T>
class Rational {
public:
  ...
friend const Rational operator*(const Rational& lhs, const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),          // same impl
                  lhs.denominator() * rhs.denominator());     // as in
}                                                             // Item 24
};
```



```
template<typename T> class Rational;                         // declare
                                                             // Rational
                                                             // template

template<typename T>                                         // declare
const Rational<T> doMultiply( const Rational<T>& lhs,        // helper
                              const Rational<T>& rhs);       // template

template<typename T>
class Rational {
public:
  ...

friend
  const Rational<T> operator*(const Rational<T>& lhs,
                              const Rational<T>& rhs)        // Have friend
  { return doMultiply(lhs, rhs); }                           // call helper
  ...
};
```

```
template<typename T>                                            // define
const Rational<T> doMultiply(const Rational<T>& lhs,            // helper
                             const Rational<T>& rhs)            // template in
{                                                               // header file,
  return Rational<T>(lhs.numerator() * rhs.numerator(),         // if necessary
                     lhs.denominator() * rhs.denominator());
}
```

```
template<typename IterT, typename DistT>      // move iter d units
void advance(IterT& iter, DistT d);           // forward; if d < 0,
                                              // move iter backward
```

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public bidirectional_iterator_tag {};
```

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (iter is a random access iterator) {
     iter += d;                                      // use iterator arithmetic
  }                                                  // for random access iters
  else {
     if (d >= 0) { while (d--) ++iter; }             // use iterative calls to
     else { while (d++) --iter; }                    // ++ or -- for other
  }                                                  // iterator categories
}
```

```
template<typename IterT>                    // template for information about
struct iterator_traits;                     // iterator types
```

```
template < ... >                              // template params elided
class deque {
public:
   class iterator {
   public:
      typedef random_access_iterator_tag iterator_category;
      ...
   };
   ...
};
```

```
template < ... >
class list {
public:
   class iterator {
   public:
      typedef bidirectional_iterator_tag iterator_category;
      ...
   };
   ...
};
```

```
// the iterator_category for type IterT is whatever IterT says it is;
// see Item 42 for info on the use of "typedef typename"
template<typename IterT>
struct iterator_traits {
  typedef typename IterT::iterator_category iterator_category;
  ...
};
```

```
template<typename T>                    // partial template specialization
struct iterator_traits<T*>              // for built-in pointer types
{
   typedef random_access_iterator_tag iterator_category;
   ...
};
```

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
   if (typeid(typename std::iterator_traits<IterT>::iterator_category) ==
      typeid(std::random_access_iterator_tag))
   ...
}
```

```
template<typename IterT, typename DistT>          // use this impl for
void doAdvance(IterT& iter, DistT d,              // random access
               std::random_access_iterator_tag)   // iterators
{
  iter += d;
}

template<typename IterT, typename DistT>          // use this impl for
void doAdvance(IterT& iter, DistT d,              // bidirectional
               std::bidirectional_iterator_tag)   // iterators
{
  if (d >= 0) { while (d--) ++iter; }
  else { while (d++) --iter; }
}

template<typename IterT, typename DistT>          // use this impl for
void doAdvance(IterT& iter, DistT d,              // input iterators
               std::input_iterator_tag)
{
  if (d < 0 ) {
    throw std::out_of_range("Negative distance");   // see below
  }
  while (d--) ++iter;
}
```

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  doAdvance(                                              // call the version
    iter, d,                                              // of doAdvance
    typename                                              // that is
      std::iterator_traits<IterT>::iterator_category()    // appropriate for
  );                                                      // iter's iterator
}                                                         // category
```

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (iter is a random access iterator) {
     iter += d;                                       // use iterator arithmetic
  }                                                   // for random access iters
  else {
     if (d >= 0) { while (d--) ++iter; }              // use iterative calls to
     else { while (d++) --iter; }                     // ++ or -- for other
  }                                                   // iterator categories
}
```

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if ( typeid(typename std::iterator_traits<IterT>::iterator_category) ==
       typeid(std::random_access_iterator_tag)) {
    iter += d;                                     // use iterator arithmetic
  }                                                // for random access iters
  else {
    if (d >= 0) { while (d--) ++iter; }            // use iterative calls to
    else { while (d++) --iter; }                   // ++ or -- for other
  }                                                // iterator categories
}
```

```
std::list<int>::iterator iter;
...
advance(iter, 10);                          // move iter 10 elements forward;
                                            // won't compile with above impl.
```

```
void advance(std::list<int>::iterator& iter, int d)
{
   if (typeid(std::iterator_traits<std::list<int>::iterator>::iterator_category) ==
       typeid(std::random_access_iterator_tag)) {
      iter += d;                                         // error! won't compile
   }
   else {
      if (d >= 0) { while (d--) ++iter; }
      else { while (d++) --iter; }
   }
}
```

```
template<unsigned n>                    // general case: the value of
struct Factorial {                      // Factorial<n> is n times the value
                                        // of Factorial<n-1>

  enum { value = n * Factorial<n-1>::value };

};

template<>                              // special case: the value of
struct Factorial<0> {                   // Factorial<0> is 1
  enum { value = 1 };

};
```

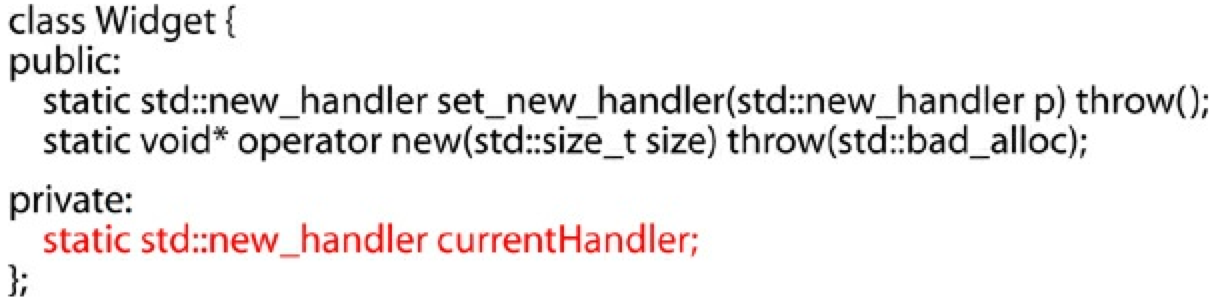
```
int main()
{
   std::cout << Factorial<5>::value;          // prints 120
   std::cout << Factorial<10>::value;         // prints 3628800
}
```

```
typedef SquareMatrix<double, 10000> BigMatrix;
BigMatrix m1, m2, m3, m4, m5;                  // create matrices and
...                                            // give them values
BigMatrix result = m1 * m2 * m3 * m4 * m5;     // compute their product
```

```
namespace std {
   typedef void (*new_handler)();
   new_handler set_new_handler(new_handler p) throw();
}
```

```
// function to call if operator new can’t allocate enough memory
void outOfMem()
{
   std::cerr << "Unable to satisfy request for memory\n";
   std::abort();
}
int main()
{
   std::set_new_handler(outOfMem);
   int *pBigDataArray = new int[100000000L];
   ...
}
```

```
class X {
public:
   static void outOfMemory();
   ...
};

class Y {
public:
   static void outOfMemory();
   ...
};

X* p1 = new X;                        // if allocation is unsuccessful,
                                      // call X::outOfMemory

Y* p2 = new Y;                        // if allocation is unsuccessful,
                                      // call Y::outOfMemory
```



```
class Widget {
public:
   static std::new_handler set_new_handler(std::new_handler p) throw();
   static void* operator new(std::size_t size) throw(std::bad_alloc);

private:
   static std::new_handler currentHandler;
};
```

```
std::new_handler Widget::currentHandler = 0;      // init to null in the class
                                                  // impl. file
```

```
std::new_handler Widget::set_new_handler(std::new_handler p) throw()
{
   std::new_handler oldHandler = currentHandler;
   currentHandler = p;
   return oldHandler;
}
```

```
class NewHandlerHolder {
public:
  explicit NewHandlerHolder(std::new_handler nh)     // acquire current
  : handler(nh) {}                                   // new-handler

  ~NewHandlerHolder()                                // release it
  { std::set_new_handler(handler); }

private:
  std::new_handler handler;                          // remember it
  NewHandlerHolder(const NewHandlerHolder&);         // prevent copying
  NewHandlerHolder&                                  // (see Item 14)
    operator=(const NewHandlerHolder&);
};
```

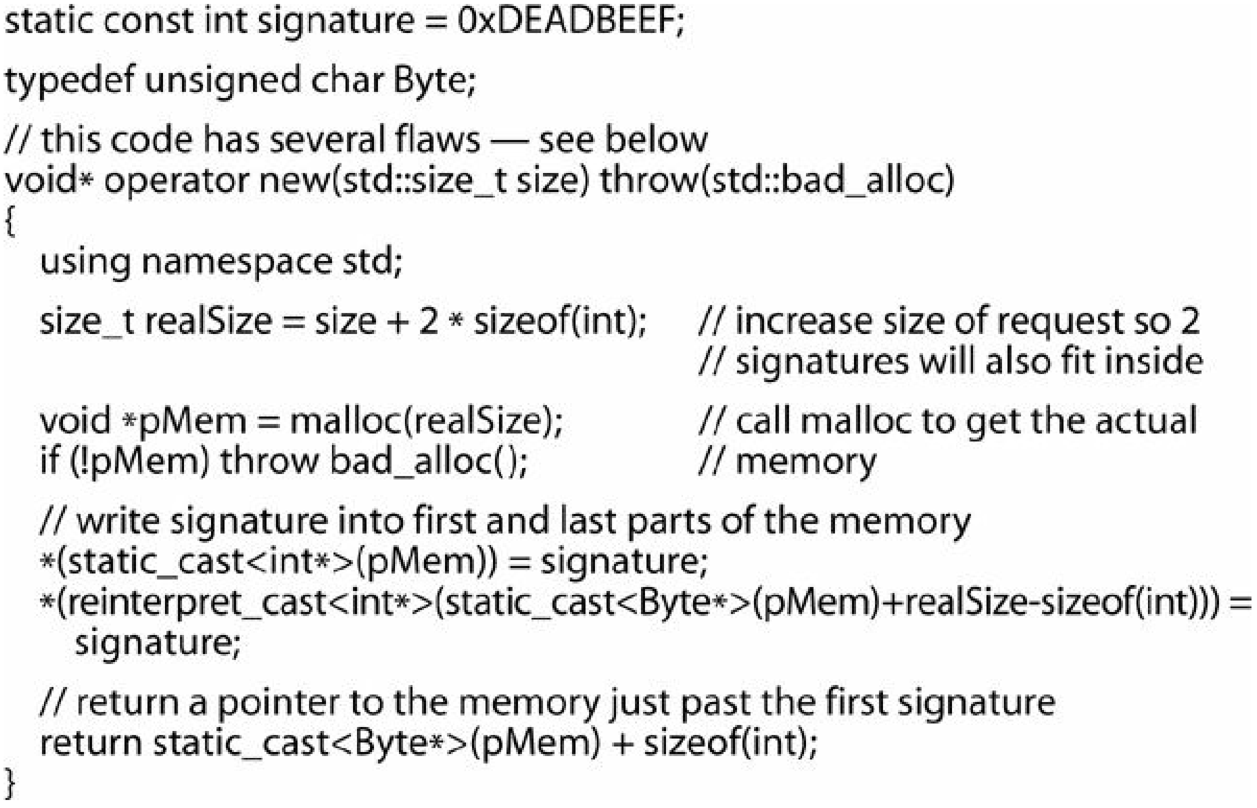
```
void* Widget::operator new(std::size_t size) throw(std::bad_alloc)
{
  NewHandlerHolder                               // install Widget's
    h(std::set_new_handler(currentHandler));     // new-handler

  return ::operator new(size);                   // allocate memory
                                                 // or throw

}                                                // restore global
                                                 // new-handler
```

```
void outOfMem();                     // decl. of func. to call if mem. alloc.
                                     // for Widget objects fails

Widget::set_new_handler(outOfMem);   // set outOfMem as Widget's
                                     // new-handling function

Widget *pw1 = new Widget;            // if memory allocation
                                     // fails, call outOfMem

std::string *ps = new std::string;   // if memory allocation fails,
                                     // call the global new-handling
                                     // function (if there is one)

Widget::set_new_handler(0);          // set the Widget-specific
                                     // new-handling function to
                                     // nothing (i.e., null)

Widget *pw2 = new Widget;            // if mem. alloc. fails, throw an
                                     // exception immediately. (There is
                                     // no new- handling function for
                                     // class Widget.)
```

```
template<typename T>                    // "mixin-style" base class for
class NewHandlerSupport {               // class-specific set_new_handler
public:                                 // support

  static std::new_handler set_new_handler(std::new_handler p) throw();
  static void* operator new(std::size_t size) throw(std::bad_alloc);

  ...                                   // other versions of op. new —
                                        // see Item 52
private:
  static std::new_handler currentHandler;
};

template<typename T>
std::new_handler
NewHandlerSupport<T>::set_new_handler(std::new_handler p) throw()
{
  std::new_handler oldHandler = currentHandler;
  currentHandler = p;
  return oldHandler;
}

template<typename T>
void* NewHandlerSupport<T>::operator new(std::size_t size)
  throw(std::bad_alloc)
{
  NewHandlerHolder h(std::set_new_handler(currentHandler));
  return ::operator new(size);
}

// this initializes each currentHandler to null
template<typename T>
std::new_handler NewHandlerSupport<T>::currentHandler = 0;
```

```
class Widget: public NewHandlerSupport<Widget> {
  ...                                     // as before, but without declarations for
};                                        // set_new_handler or operator new
```

```
class Widget { ... };
Widget *pw1 = new Widget;                  // throws bad_alloc if
                                           // allocation fails
if (pw1 == 0) ...                          // this test must fail


Widget *pw2 = new (std::nothrow) Widget;   // returns 0 if allocation for
                                           // the Widget fails
if (pw2 == 0) ...                          // this test may succeed
```



```
static const int signature = 0xDEADBEEF;

typedef unsigned char Byte;

// this code has several flaws — see below
void* operator new(std::size_t size) throw(std::bad_alloc)
{
  using namespace std;

  size_t realSize = size + 2 * sizeof(int);     // increase size of request so 2
                                                // signatures will also fit inside

  void *pMem = malloc(realSize);                // call malloc to get the actual
  if (!pMem) throw bad_alloc();                 // memory

  // write signature into first and last parts of the memory
  *(static_cast<int*>(pMem)) = signature;
  *(reinterpret_cast<int*>(static_cast<Byte*>(pMem)+realSize-sizeof(int))) =
    signature;

  // return a pointer to the memory just past the first signature
  return static_cast<Byte*>(pMem) + sizeof(int);
}
```

```
void* operator new(std::size_t size) throw(std::bad_alloc)
{                                                  // your operator new might
  using namespace std;                             // take additional params
  if (size == 0) {                                 // handle 0-byte requests
    size = 1;                                      // by treating them as
  }                                                // 1-byte requests

  while (true) {
    attempt to allocate size bytes;

    if (the allocation was successful)
      return (a pointer to the memory);

    // allocation was unsuccessful; find out what the
    // current new-handling function is (see below)
    new_handler globalHandler = set_new_handler(0);
    set_new_handler(globalHandler);

    if (globalHandler) (*globalHandler)();
    else throw std::bad_alloc();
  }
}
```

```
class Base {
public:
  static void* operator new(std::size_t size) throw(std::bad_alloc);
  ...
};

class Derived: public Base                        // Derived doesn't declare
{ ... };                                          // operator new

Derived *p = new Derived;                         // calls Base::operator new!
```

```
void* Base::operator new(std::size_t size) throw(std::bad_alloc)
{
  if (size != sizeof(Base))                   // if size is “wrong,”
    return ::operator new(size);              // have standard operator
                                              // new handle the request

  ...                                         // otherwise handle
                                              // the request here
}
```

```
void operator delete(void *rawMemory) throw()
{
  if (rawMemory == 0) return;            // do nothing if the null
                                         // pointer is being deleted
  deallocate the memory pointed to by rawMemory;
}
```

```
class Base {                                    // same as before, but now
public:                                         // operator delete is declared

  static void* operator new(std::size_t size) throw(std::bad_alloc);
  static void operator delete(void *rawMemory, std::size_t size) throw();
  ...
};

void Base::operator delete(void *rawMemory, std::size_t size) throw()
{
  if (rawMemory == 0) return;                   // check for null pointer

  if (size != sizeof(Base)) {                   // if size is "wrong,"
    ::operator delete(rawMemory);               // have standard operator
    return;                                     // delete handle the request
  }

  deallocate the memory pointed to by rawMemory;
  return;
}
```

```
void operator delete(void *rawMemory) throw();     // normal signature
                                                   // at global scope

void operator delete(void *rawMemory,              // typical normal
                     std::size_t size) throw();    // signature at class
                                                   // scope
```

```
class Widget {
public:
  ...
  static void* operator new(std::size_t size,                     // non-normal
                             std::ostream& logStream)             // form of new
    throw(std::bad_alloc);
  static void operator delete(void *pMemory,                      // normal class-
                              std::size_t size) throw();          // specific form
                                                                  // of delete
  ...
};
```

```
void* operator new(std::size_t, void *pMemory) throw();       // “placement
                                                               // new”
```

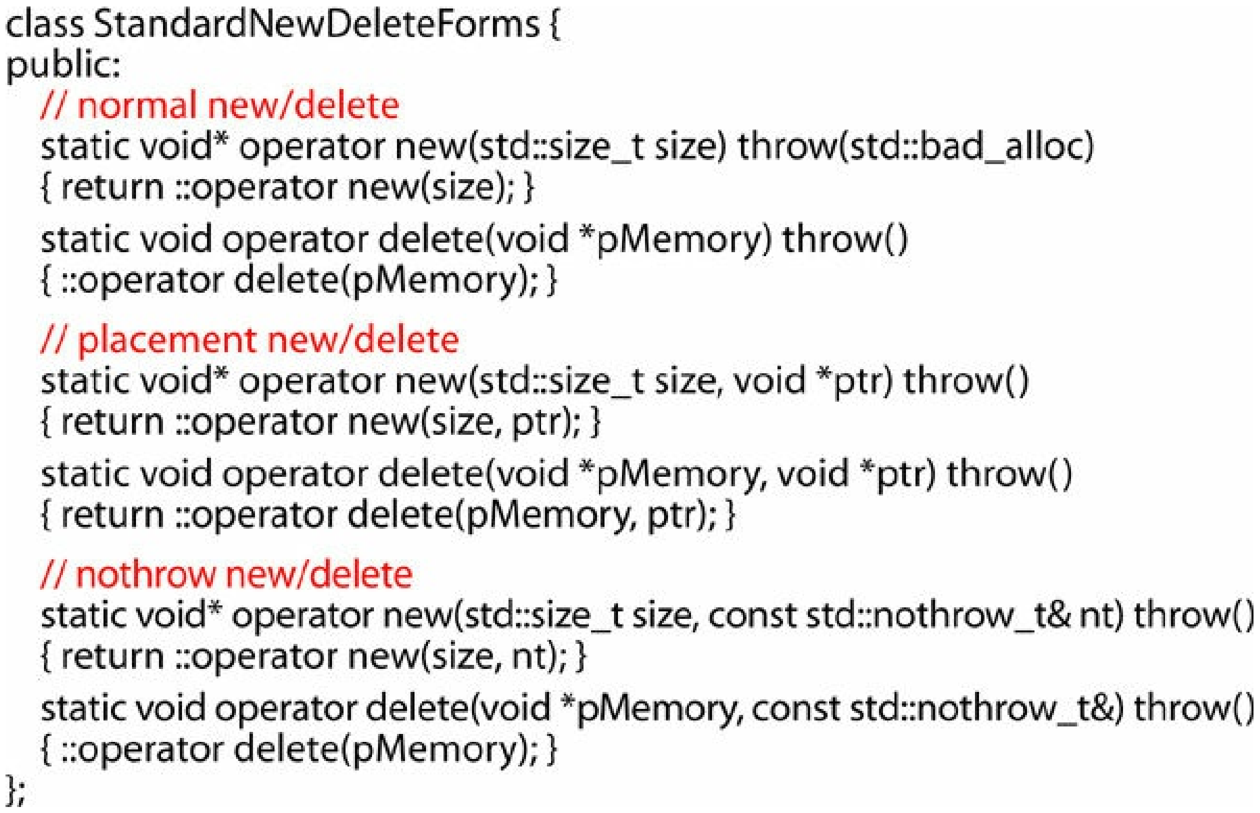
```
Widget *pw = new (std::cerr) Widget;   // call operator new, passing cerr as
                                       // the ostream; this leaks memory
                                       //  if the Widget constructor throws
```

```
class Widget {
public:
   ...
   static void* operator new(std::size_t size, std::ostream& logStream)
      throw(std::bad_alloc);
   static void operator delete(void *pMemory) throw();

   static void operator delete(void *pMemory, std::ostream& logStream)
      throw();

   ...
};
```

```
class Base {
public:
  ...
  static void* operator new(std::size_t size,                    // this new hides
                            std::ostream& logStream)             // the normal
    throw(std::bad_alloc);                                       // global forms
  ...
};

Base *pb = new Base;                          // error! the normal form of
                                              // operator new is hidden

Base *pb = new (std::cerr) Base;              // fine, calls Base's
                                              // placement new
```

```
class Derived: public Base {                          // inherits from Base above
public:
  ...
  static void* operator new(std::size_t size)        // redeclares the normal
    throw(std::bad_alloc);                            // form of new
  ...
};

Derived *pd = new (std::clog) Derived;                // error! Base's placement
                                                      // new is hidden

Derived *pd = new Derived;                            // fine, calls Derived's
                                                      // operator new
```

```
void* operator new(std::size_t) throw(std::bad_alloc);      // normal new
void* operator new(std::size_t, void*) throw();             // placement new
void* operator new(std::size_t,                             // nothrow new —
                   const std::nothrow_t&) throw();          // see Item 49
```



```
class StandardNewDeleteForms {
public:
  // normal new/delete
  static void* operator new(std::size_t size) throw(std::bad_alloc)
  { return ::operator new(size); }
  static void operator delete(void *pMemory) throw()
  { ::operator delete(pMemory); }
  // placement new/delete
  static void* operator new(std::size_t size, void *ptr) throw()
  { return ::operator new(size, ptr); }
  static void operator delete(void *pMemory, void *ptr) throw()
  { return ::operator delete(pMemory, ptr); }
  // nothrow new/delete
  static void* operator new(std::size_t size, const std::nothrow_t& nt) throw()
  { return ::operator new(size, nt); }
  static void operator delete(void *pMemory, const std::nothrow_t&) throw()
  { ::operator delete(pMemory); }
};
```

```
class Widget: public StandardNewDeleteForms {           // inherit std forms
public:
  using StandardNewDeleteForms::operator new;           // make those
  using StandardNewDeleteForms::operator delete;        // forms visible

  static void* operator new(std::size_t size,           // add a custom
                            std::ostream& logStream)    // placement new
    throw(std::bad_alloc);

  static void operator delete(void *pMemory,            // add the corres-
                              std::ostream& logStream)  // ponding place-
    throw();                                            // ment delete
  ...
};
```

```
void registerCallback(std::string func(int));     // param type is a function
                                                  // taking an int and
                                                  // returning a string
```

```
void registerCallback(std::string (int));          // same as above; param
                                                   // name is omitted
```

```
void registerCallback(std::tr1::function<std::string (int)> func);
                                                  // the param “func” will
                                                  // take any callable entity
                                                  // with a sig consistent
                                                  // with “std::string (int)”
```

```
namespace std {
   namespace tr1 = ::boost;                    // namespace std::tr1 is an alias
}                                              // for namespace boost
```

```
using namespace boost::lambda;                 // make boost::lambda
                                               // functionality visible

std::vector<int> v;

...

std::for_each(v.begin(), v.end(),              // for each element x in
              std::cout << _1 * 2 + 10 << "\n");  // v, print x*2+10;
                                               // "_1" is the Lambda
                                               // library's placeholder
                                               // for the current element
// create a list-like compile-time container of three types (float,
// double, and long double) and call the container “floats”
typedef boost::mpl::list<float, double, long double> floats;

// create a new compile-time list of types consisting of the types in
// “floats” plus “int” inserted at the front; call the new container “types”
typedef boost::mpl::push_front<floats, int>::type types;
```

```
void f(int);
void f(bool);

int x, y;
...
f( x < y );                              // calls f(int), but it
                                         // should call f(bool)
```

```
GameObject *pgo =                 // static type of pgo is
  new SpaceShip;                  // GameObject*, dynamic
                                  // type is SpaceShip*

Asteroid *pa = new Asteroid;      // static type of pa is
                                  // Asteroid*. So is its
                                  // dynamic type

pgo = pa;                         // static type of pgo is
                                  // still (and always)
                                  // GameObject*. Its
                                  // dynamic type is now
                                  // Asteroid*

GameObject& rgo = *pa;            // static type of rgo is
                                  // GameObject, dynamic
                                  // type is Asteroid
```

```
class Widget { ... };             // some class — it doesn't
                                  // matter what it is

Widget *pw = new Widget;          // dynamically allocate a
                                  // Widget object

...                               // assume pw is never
                                  // deleted
```

```
char *pc = 0;                   // set pointer to null
char& rc = *pc;                 // make reference refer to
                                // dereferenced null pointer
```

```
string& rs;                     // error! References must
                                // be initialized
string s("xyzzy");
string& rs = s;                 // okay, rs refers to s
```

```
string *ps;                         // uninitialized pointer:
                                    // valid but risky
```

```
void printDouble(const double& rd)
{
   cout << rd;                        // no need to test rd; it
}                                     // must refer to a double
```

```
void printDouble(const double *pd)
{
  if (pd) {                    // check for null pointer
    cout << *pd;
  }
}
```

```
string s1("Nancy");
string s2("Clancy");

string& rs = s1;                // rs refers to s1

string *ps = &s1;               // ps points to s1

rs = s2;                        // rs still refers to s1,
                                // but s1’s value is now
                                // "Clancy"

ps = &s2;                       // ps now points to s2;
                                // s1 is unchanged
```

```
vector<int> v(10);  // create an int vector of size 10;
                    // vector is a template in the
                    // standard C++ library (see Item 35)

v[5] = 10;          // the target of this assignment is
                    // the return value of operator[]
```

```
int firstNumber, secondNumber;
...
double result = ((double)firstNumber)/secondNumber;
```

```
class Widget { ... };
class SpecialWidget: public Widget { ... };

void update(SpecialWidget *psw);

SpecialWidget sw;               // sw is a non-const object,
const SpecialWidget& csw = sw;  // but csw is a reference to
                                // it as a const object

update(&csw);         // error! can't pass a const
                      // SpecialWidget* to a function
                      // taking a SpecialWidget*

update(const_cast<SpecialWidget*>(&csw));
                      // fine, the constness of &csw is
                      // explicitly cast away (and
                      // csw — and sw — may now be
                      // changed inside update)

update((SpecialWidget*)&csw);
                      // same as above, but using a
                      // harder-to-recognize C-style cast

Widget *pw = new SpecialWidget;

update(pw);           // error! pw's type is Widget*, but
                      // update takes a SpecialWidget*

update(const_cast<SpecialWidget*>(pw));
                      // error! const_cast can be used only
                      // to affect constness or volatileness,
                      // never to cast down the inheritance
                      // hierarchy
```

```
Widget *pw;

...

update(dynamic_cast<SpecialWidget*>(pw));
                        // fine, passes to update a pointer
                        // to the SpecialWidget pw points to
                        // if pw really points to one,
                        // otherwise passes the null pointer

void updateViaRef(SpecialWidget& rsw);

updateViaRef(dynamic_cast<SpecialWidget&>(*pw));
                        // fine, passes to updateViaRef the
                        // SpecialWidget pw points to if pw
                        // really points to one, otherwise
                        // throws an exception
```

```
int firstNumber, secondNumber;
...
double result = dynamic_cast<double>(firstNumber)/secondNumber;
                        // error! int has no virtual functions

const SpecialWidget sw;
...
update(dynamic_cast<SpecialWidget*>(&sw));
                        // error! dynamic_cast can't cast
                        // away constness
```

```
typedef void (*FuncPtr)();      // a FuncPtr is a pointer
                                // to a function taking no
                                // args and returning void

FuncPtr funcPtrArray[10];       // funcPtrArray is an array
                                // of 10 FuncPtrs
```

```
funcPtrArray[0] =                      // this compiles
  reinterpret_cast<FuncPtr>(&doSomething);
```

```
#define static_cast(TYPE,EXPR)        ((TYPE)(EXPR))
#define const_cast(TYPE,EXPR)         ((TYPE)(EXPR))
#define reinterpret_cast(TYPE,EXPR) ((TYPE)(EXPR))
```

```
double result = static_cast(double, firstNumber)/secondNumber;
update(const_cast(SpecialWidget*, &sw));
funcPtrArray[0] = reinterpret_cast(FuncPtr, &doSomething);
```

```
void printBSTArray(ostream& s,
                   const BST array[],
                   int numElements)
{
  for (int i = 0; i < numElements; ++i) {
    s << array[i];                 // this assumes an
  }                                // operator<< is defined
}                                  // for BST objects
```

```
BST BSTArray[10];
...
printBSTArray(cout, BSTArray, 10);           // works fine
```

```
BalancedBST bBSTArray[10];
...
printBSTArray(cout, bBSTArray, 10);           // works fine?
```

```
for (int i = 0; i < numElements; ++i) {
  s << array[i];
}
```

```
// delete an array, but first log a message about its
// deletion
void deleteArray(ostream& logStream, BST array[])
{
  logStream << "Deleting array at address "
            << static_cast<void*>(array) << ’\n’;

  delete [] array;
}

BalancedBST *balTreeArray =        // create a BalancedBST
  new BalancedBST[50];             // array

...

deleteArray(cout, balTreeArray);   // log its deletion
```

```
// destruct the objects in *array in the inverse order
// in which they were constructed
for (int i = the number of elements in the array - 1;
     i >= 0;
     --i)
  {
   array[i].BST::~BST();       // call array[i]’s
  }                            // destructor
```

```
class EquipmentPiece {
public:
  EquipmentPiece(int IDNumber);
  ...
};
```

```
EquipmentPiece bestPieces[10];       // error! No way to call
                                     // EquipmentPiece ctors

EquipmentPiece *bestPieces =
  new EquipmentPiece[10];            // error! same problem
```

```
int ID1, ID2, ID3, ..., ID10;      // variables to hold
                                   // equipment ID numbers
...

EquipmentPiece bestPieces[] = {    // fine, ctor arguments
  EquipmentPiece(ID1),             // are provided
  EquipmentPiece(ID2),
  EquipmentPiece(ID3),
  ...,
  EquipmentPiece(ID10)
};
```

```
typedef EquipmentPiece* PEP;       // a PEP is a pointer to
                                   // an EquipmentPiece
PEP bestPieces[10];                // fine, no ctors called
PEP *bestPieces = new PEP[10];     // also fine
```

```
for (int i = 0; i < 10; ++i)
 bestPieces[i] = new EquipmentPiece( ID Number );
```

```
// allocate enough raw memory for an array of 10
// EquipmentPiece objects; see Item 8 for details on
// the operator new[] function
void *rawMemory =
  operator new[](10*sizeof(EquipmentPiece));

// make bestPieces point to it so it can be treated as an
// EquipmentPiece array
EquipmentPiece *bestPieces =
  static_cast<EquipmentPiece*>(rawMemory);

// construct the EquipmentPiece objects in the memory
// using "placement new" (see Item 8)
for (int i = 0; i < 10; ++i)
  new (bestPieces+i) EquipmentPiece( ID Number );
```

```
// destruct the objects in bestPieces in the inverse
// order in which they were constructed
for (int i = 9; i >= 0; --i)
  bestPieces[i].~EquipmentPiece();

// deallocate the raw memory
operator delete[](rawMemory);
```

```
delete [] bestPieces;              // undefined! bestPieces
                                   // didn't come from the new
                                   // operator
```

```
template<class T>
class Array {
public:
  Array(int size);
  ...

private:
  T *data;
};

template<class T>
Array<T>::Array(int size)
{
  data = new T[size];             // calls T::T() for each
  ...                             // element of the array
}
```

```
class EquipmentPiece {
public:
  EquipmentPiece(int IDNumber = UNSPECIFIED);
  ...

private:
  static const int UNSPECIFIED;   // magic ID number value
                                  // meaning no ID was
};                                // specified
```

```
class Name {                             // for names of things
public:
  Name(const string& s);                 // converts string to
                                         // Name
  ...

};

class Rational {                         // for rational numbers
public:
  Rational(int numerator = 0,            // converts int to
           int denominator = 1);         // Rational
  ...

};
```

```
class Rational {
public:
  ...
  operator double() const;        // converts Rational to
};                                // double
```

```
Rational r(1, 2);                // r has the value 1/2
double d = 0.5 * r;              // converts r to a double,
                                 // then does multiplication
```

```
class Rational {
public:
  ...
  double asDouble() const;        // converts Rational
};                                // to double
```

```
Rational r(1, 2);
cout << r;                      // error! No operator<<
                                // for Rationals
cout << r.asDouble();           // fine, prints r as a
                                // double
```

```
template<class T>
class Array {
public:
  Array(int lowBound, int highBound);
  Array(int size);

  T& operator[](int index);

  ...

};
```

```
bool operator==( const Array<int>& lhs,
                 const Array<int>& rhs);

Array<int> a(10);
Array<int> b(10);

...

for (int i = 0; i < 10; ++i)
  if (a == b[i]) {              // oops! "a" should be "a[i]"
    do something for when
    a[i] and b[i] are equal;
  }
  else {
    do something for when they're not;
  }
```

```
template<class T>
class Array {
public:
  ...
  explicit Array(int size);      // note use of "explicit"
  ...
};

Array<int> a(10);                // okay, explicit ctors can
                                 // be used as usual for
                                 // object construction

Array<int> b(10);                // also okay

if (a == b[i]) ...               // error! no way to
                                 // implicitly convert
                                 // int to Array<int>

if (a == Array<int>(b[i])) ...   // okay, the conversion
                                 // from int to Array<int> is
                                 // explicit (but the logic of
                                 // the code is suspect)

if (a == static_cast< Array<int> >(b[i])) ...
                                 // equally okay, equally
                                 // suspect

if (a == (Array<int>)b[i]) ...   // C-style casts are also
                                 // okay, but the logic of
                                 // the code is still suspect
```

```
template<class T>
class Array {
public:

  class ArraySize {                 // this class is new
  public:
    ArraySize(int numElements): theSize(numElements) {}
    int size() const { return theSize; }

  private:
    int theSize;
  };

  Array(int lowBound, int highBound);
  Array(ArraySize size);            // note new declaration

  ...

};
```

```
bool operator==( const Array<int>& lhs,
                 const Array<int>& rhs);

Array<int> a(10);
Array<int> b(10);

...

for (int i = 0; i < 10; ++i)
  if (a == b[i]) ...           // oops! "a" should be "a[i]";
                               // this is now an error
```

```
class UPInt {                   // "unlimited precision int"
public:
  UPInt& operator++();          // prefix ++
  const UPInt operator++(int);  // postfix ++

  UPInt& operator--();          // prefix --
  const UPInt operator--(int);  // postfix --

  UPInt& operator+=(int);       // a += operator for UPInts
                                // and ints

  ...

};

UPInt i;

++i;                            // calls i.operator++();
i++;                            // calls i.operator++(0);

--i;                            // calls i.operator--();
i--;                            // calls i.operator--(0);
```

```
// prefix form: increment and fetch
UPInt& UPInt::operator++()
{
  *this += 1;                       // increment
  return *this;                     // fetch
}

// postfix form: fetch and increment
const UPInt UPInt::operator++(int)
{
  const UPInt oldValue = *this;     // fetch
  ++(*this);                        // increment

  return oldValue;                  // return what was
}                                   // fetched
```

```
UPInt i;
i++++;                                  // apply postfix increment
                                        // twice
```

```
int rangeCheck(int index)
{
  if ((index < lowerBound) || (index > upperBound)) ...

  ...

}
```

```
if (expression1.operator&&(expression2)) ...
                                     // when operator&& is a
                                     // member function
if (operator&&(expression1, expression2)) ...
                                     // when operator&& is a
                                     // global function
// reverse string s in place
void reverse(char s[])
{

  for (int i = 0, j = strlen(s)-1;
       i < j;
       ++i, --j)                // aha! the comma operator!
  {
   int c = s[i];
   s[i] = s[j];
   s[j] = c;
  }
}
```

`.`	`.*`	`::`	`?:`
`new`	`delete`	`sizeof`	`typeid`
`static_cast`	`dynamic_cast`	`const_cast`	`reinterpret_cast`

```
void *memory =                            // get raw memory
  operator new(sizeof(string));           // for a string
                                          // object

call string::string("Memory Management")  // initialize the
on *memory;                               // object in the
                                          // memory

string *ps =                              // make ps point to
  static_cast<string*>(memory);           // the new object
```

```
class Widget {
public:
  Widget(int widgetSize);
  ...
};

Widget * constructWidgetInBuffer(void *buffer,
                                 int widgetSize)
{
  return new (buffer) Widget(widgetSize);
}
```

```
void * operator new(size_t, void *location)
{
  return location;
}
```

```
string *ps;
...
delete ps;                              // use the delete operator
```

```
ps->~string();                   // call the object’s dtor
operator delete(ps);             // deallocate the memory
                                 // the object occupied
```

```
void *buffer =                      // allocate enough
  operator new(50*sizeof(char));    // memory to hold 50
                                    // chars; call no ctors

...

operator delete(buffer);            // deallocate the memory;
                                    // call no dtors
// functions for allocating and deallocating memory in
// shared memory
void * mallocShared(size_t size);
void freeShared(void *memory);

void *sharedMemory = mallocShared(sizeof(Widget));

Widget *pw =                                       // as above,
  constructWidgetInBuffer( sharedMemory, 10);      // placement
                                                   // new is used

...

delete pw;            // undefined! sharedMemory came from
                      // mallocShared, not operator new

pw->~Widget();        // fine, destructs the Widget pointed to
                      // by pw, but doesn't deallocate the
                      // memory containing the Widget

freeShared(pw);       // fine, deallocates the memory pointed
                      // to by pw, but calls no destructor
```

```
string *ps = new string[10];    // allocate an array of
                                // objects
```

```
string *ps =             // call operator new[] to allocate
  new string[10];        // memory for 10 string objects,
                         // then call the default string
                         // ctor for each array element
```

```
delete [] ps;                 // call the string dtor for each
                              // array element, then call
                              // operator delete[] to
                              // deallocate the array's memory
```

```
class ALA {
public:
  virtual void processAdoption() = 0;
  ...
};

class Puppy: public ALA {
public:
  virtual void processAdoption();
  ...
};

class Kitten: public ALA {
public:
  virtual void processAdoption();
  ...
};
```

```
// read animal information from s, then return a pointer
// to a newly allocated object of the appropriate type
ALA * readALA(istream& s);
```

```
void processAdoptions(istream& dataSource)
{
  while (dataSource) {                 // while there's data
    ALA *pa = readALA(dataSource);     // get next animal
    pa->processAdoption();             // process adoption
    delete pa;                         // delete object that
  }                                    // readALA returned
}
```

```
void processAdoptions(istream& dataSource)
{
  while (dataSource) {
    ALA *pa = readALA(dataSource);

    try {
      pa->processAdoption();
    }
    catch (...) {              // catch all exceptions

      delete pa;               // avoid resource leak when an
                               // exception is thrown

      throw;                   // propagate exception to caller
    }

    delete pa;                 // avoid resource leak when no
  }                            // exception is thrown
}
```

```
template<class T>
class auto_ptr {
public:
  auto_ptr(T *p = 0): ptr(p) {}    // save ptr to object
  ~auto_ptr() { delete ptr; }      // delete ptr to object

private:
  T *ptr;                          // raw ptr to object
};
```

```
void processAdoptions(istream& dataSource)
{
  while (dataSource) {
    auto_ptr<ALA> pa(readALA(dataSource));
    pa->processAdoption();
  }
}
```

```
// this function may leak resources if an exception
// is thrown
void displayInfo(const Information& info)
{
  WINDOW_HANDLE w(createWindow());
  display info in window corresponding to w;
  destroyWindow(w);
}
```

```
// class for acquiring and releasing a window handle
class WindowHandle {
public:
  WindowHandle(WINDOW_HANDLE handle): w(handle) {}
  ~WindowHandle() { destroyWindow(w); }

  operator WINDOW_HANDLE() { return w; }     // see below

private:
  WINDOW_HANDLE w;

  // The following functions are declared private to prevent
  // multiple copies of a WINDOW_HANDLE from being created.
  // See Item 28 for a discussion of a more flexible approach.
  WindowHandle(const WindowHandle&);
  WindowHandle& operator=(const WindowHandle&);
};
```

```
// this function avoids leaking resources if an
// exception is thrown
void displayInfo(const Information& info)
{
  WindowHandle w(createWindow());

  display info in window corresponding to w;

}
```

```
class Image {                  // for image data
public:
  Image(const string& imageDataFileName);
  ...
};

class AudioClip {              // for audio data
public:
  AudioClip(const string& audioDataFileName);
  ...
};

class PhoneNumber { ... };     // for holding phone numbers

class BookEntry {              // for each entry in the
public:                        // address book

  BookEntry(const string& name,
            const string& address = "",
            const string& imageFileName = "",
            const string& audioClipFileName = "");
  ~BookEntry();

  // phone numbers are added via this function
  void addPhoneNumber(const PhoneNumber& number);
  ...

private:
  string theName;              // person's name
  string theAddress;           // their address
  list<PhoneNumber> thePhones; // their phone numbers
  Image *theImage;             // their image
  AudioClip *theAudioClip;     // an audio clip from them
};
```

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(0), theAudioClip(0)
{
  if (imageFileName != "") {
    theImage = new Image(imageFileName);
  }

  if (audioClipFileName != "") {
    theAudioClip = new AudioClip(audioClipFileName);
  }
}

BookEntry::~BookEntry()
{
  delete theImage;
  delete theAudioClip;
}
```

```
if (audioClipFileName != "") {
  theAudioClip = new AudioClip(audioClipFileName);
}
```

```
void testBookEntryClass()
{
  BookEntry b("Addison-Wesley Publishing Company",
              "One Jacob Way, Reading, MA 01867");

  ...

}
```

```
void testBookEntryClass()
{
  BookEntry *pb = 0;

  try {
    pb = new BookEntry( "Addison-Wesley Publishing Company",
                        "One Jacob Way, Reading, MA 01867");
    ...
  }
  catch (...) {                  // catch all exceptions

    delete pb;                   // delete pb when an
                                 // exception is thrown

    throw;                       // propagate exception to
  }                              // caller

  delete pb;                     // delete pb normally
}
```

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(0), theAudioClip(0)
{
  try {                            // this try block is new
    if (imageFileName != "") {
      theImage = new Image(imageFileName);
    }

    if (audioClipFileName != "") {
      theAudioClip = new AudioClip(audioClipFileName);
    }
  }
  catch (...) {                    // catch any exception

    delete theImage;               // perform necessary
    delete theAudioClip;           // cleanup actions

    throw;                         // propagate the exception
  }
}
```

```
class BookEntry {
public:
  ...                               // as before

private:
  ...
  void cleanup();                   // common cleanup statements
};

void BookEntry::cleanup()
{
  delete theImage;
  delete theAudioClip;
}

BookEntry::BookEntry( const string& name,
                      const string& address,
                      const string& imageFileName,
                      const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(0), theAudioClip(0)
{
  try {
    ...                             // as before
  }
  catch (...) {
    cleanup();                      // release resources
    throw;                          // propagate exception
  }
}

BookEntry::~BookEntry()
{
  cleanup();
}
```

```
class BookEntry {
public:
  ...                                        // as above

private:
  ...
  Image * const theImage;                    // pointers are now
  AudioClip * const theAudioClip;            // const
};
```

```
// an implementation that may leak resources if an
// exception is thrown
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(imageFileName != ""
           ? new Image(imageFileName)
           : 0),
  theAudioClip(audioClipFileName != ""
               ? new AudioClip(audioClipFileName)
               : 0)
{}
```

```
class BookEntry {
public:
  ...                                 // as above

private:
  ...                                 // data members as above

  Image * initImage(const string& imageFileName);
  AudioClip * initAudioClip(const string&
                            audioClipFileName);
};

BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(initImage(imageFileName)),
  theAudioClip(initAudioClip(audioClipFileName))
{}
// theImage is initialized first, so there is no need to
// worry about a resource leak if this initialization
// fails. This function therefore handles no exceptions
Image * BookEntry::initImage(const string& imageFileName)
{
  if (imageFileName != "") return new Image(imageFileName);
  else return 0;
}

// theAudioClip is initialized second, so it must make
// sure theImage's resources are released if an exception
// is thrown during initialization of theAudioClip. That's
// why this function uses try...catch.
```

```
AudioClip * BookEntry::initAudioClip(const string&
                                     audioClipFileName)
{
  try {
    if (audioClipFileName != "") {
      return new AudioClip(audioClipFileName);
    }
    else return 0;
  }
  catch (...) {
    delete theImage;
    throw;
  }
}
```

```
class BookEntry {
public:
  ...                                     // as above

private:
  ...
  const auto_ptr<Image> theImage;         // these are now
  const auto_ptr<AudioClip> theAudioClip; // auto_ptr objects
};
```

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(imageFileName != ""
           ? new Image(imageFileName)
           : 0),
  theAudioClip(audioClipFileName != ""
               ? new AudioClip(audioClipFileName)
               : 0)
{}
```

```
BookEntry::~BookEntry()
{}                                  // nothing to do!
```

```
class Session {
public:
  Session();
  ~Session();
  ...
private:
  static void logCreation(Session *objAddr);
  static void logDestruction(Session *objAddr);
};
```

```
Session::~Session()
{
  try {
    logDestruction(this);
  }
  catch (...) {
    cerr << "Unable to log destruction of Session object "
          << "at address "
          << this
          << ".\n";
  }
}
```

```
Session::Session()                  // to keep things simple,
{                                   // this ctor handles no
                                    // exceptions
  logCreation(this);
  startTransaction();               // start DB transaction
}

Session::~Session()
{
  logDestruction(this);
  endTransaction();                 // end DB transaction
}
```

```
class Widget { ... };              // some class; it makes no
                                   // difference what it is

void f1(Widget w);                 // all these functions
void f2(Widget& w);                // take parameters of
void f3(const Widget& w);          // type Widget, Widget&, or
void f4(Widget *pw);               // Widget*
void f5(const Widget *pw);

catch (Widget w) ...               // all these catch clauses
catch (Widget& w) ...              // catch exceptions of
catch (const Widget& w) ...        // type Widget, Widget&, or
catch (Widget *pw) ...             // Widget*
catch (const Widget *pw) ...
```

```
// function to read the value of a Widget from a stream
istream operator>>(istream& s, Widget& w);

void passAndThrowWidget()
{
  Widget localWidget;

  cin >> localWidget;    // pass localWidget to operator>>

  throw localWidget;     // throw localWidget as an exception
}
```

```
void passAndThrowWidget()
{
  static Widget localWidget;    // this is now static; it
                                // will exist until the
                                // end of the program
  cin >> localWidget;           // this works as before

  throw localWidget;            // a copy of localWidget is
}                               // still made and thrown
```

```
class Widget { ... };
class SpecialWidget: public Widget { ... };

void passAndThrowWidget()
{
  SpecialWidget localSpecialWidget;

  ...

  Widget& rw = localSpecialWidget;  // rw refers to a
                                    // SpecialWidget

  throw rw;                         // this throws an
                                    // exception of type
}                                   // Widget!
```

```
catch (Widget& w)                // catch Widget exceptions
{
  ...                            // handle the exception

  throw;                         // rethrow the exception so it
}                                // continues to propagate

catch (Widget& w)                // catch Widget exceptions
{
  ...                            // handle the exception

  throw w;                       // propagate a copy of the
}                                // caught exception
```

```
catch (Widget w) ...              // catch exception by value

catch (Widget& w) ...             // catch exception by
                                  // reference

catch (const Widget& w) ...       // catch exception by
                                  // reference-to-const
```

```
catch (Widget& w) ...          // catch by reference
catch (const Widget& w) ...    // also catch by reference
```

```
void f(int value)
{
  try {
    if (someFunction()) {         // if someFunction() returns
      throw value;                // true, throw an int
    }
    ...
  }
  catch (double d) {              // handle exceptions of
    ...                           // type double here
  }

  ...

}
```

```
catch (runtime_error) ...        // can catch errors of type
catch (runtime_error&) ...       // runtime_error,
catch (const runtime_error&) ... // range_error, or
                                 // overflow_error

catch (runtime_error*) ...       // can catch errors of type
catch (const runtime_error*) ... // runtime_error*,
                                 // range_error*, or
                                 // overflow_error*
```

```
catch (const void*) ...        // catches any exception
                               // that’s a pointer
```

```
try {
  ...
}
catch (logic_error& ex) {       // this block will catch
  ...                           // all logic_error
}                               // exceptions, even those
                                // of derived types

catch (invalid_argument& ex) {  // this block can never be
  ...                           // executed, because all
}                               // invalid_argument
                                // exceptions will be caught
                                // by the clause above
```

```
try {
  ...
}
catch (invalid_argument& ex) { // handle invalid_argument
  ...                          // exceptions here
}
catch (logic_error& ex) {      // handle all other
  ...                          // logic_errors here
}
```

```
class exception { ... };          // from the standard C++
                                  // library exception
                                  // hierarchy (see Item 12)

void someFunction()
{
  static exception ex;            // exception object

  ...

  throw &ex;                      // throw a pointer to ex

  ...

}

void doSomething()
{
  try {
    someFunction();               // may throw an exception*
  }
  catch (exception *ex) {         // catches the exception*;
    ...                           // no object is copied
  }
}
```

```
void someFunction()
{
  exception ex;                    // local exception object;
                                   // will be destroyed when
                                   // this function's scope is
  ...                              // exited

  throw &ex;                       // throw a pointer to an
  ...                              // object that's about to
}                                  // be destroyed
```

```
void someFunction()
{
  ...
  throw new exception;    // throw a pointer to a new heap-
  ...                     // based object (and hope that
}                         // operator new — see Item 8 —
                          // doesn't itself throw an
                          // exception!)
```

```
class exception {               // as above, this is a
public:                         // standard exception class

  virtual const char * what() const throw();
                                // returns a brief descrip.
  ...                           // of the exception (see
                                // Item 14 for info about
};                              // the "throw()" at the
                                // end of the declaration)

class runtime_error:            // also from the standard
  public exception { ... };     // C++ exception hierarchy

class Validation_error:         // this is a class added by
  public runtime_error {        // a client
public:
  virtual const char * what() const throw();
                                // this is a redefinition
  ...                           // of the function declared
};                              // in class exception above

void someFunction()             // may throw a validation
{                               // exception
  ...

  if (a validation test fails) {
    throw Validation_error();
  }

  ...

}

void doSomething()
{
  try {
    someFunction();                   // may throw a validation
  }                                   // exception

  catch (exception ex) {              // catches all exceptions
                                      // in or derived from
                                      // the standard hierarchy

    cerr << ex.what();                // calls exception::what(),
    ...                               // never
  }                                   // Validation_error::what()
}
```

```
void someFunction()             // nothing changes in this
{                               // function
  ...

  if (a validation test fails) {
    throw Validation_error();
  }

  ...

}
void doSomething()
{
  try {
    someFunction();             // no change here
  }
  catch (exception& ex) {       // here we catch by reference
                                // instead of by value

    cerr << ex.what();          // now calls
                                // Validation_error::what(),
    ...                         // not exception::what()
  }
}
```

```
void f2() throw(int)
{
  ...
  f1();                        // legal even though f1 might throw
                               // something besides an int
  ...
}
```

```
// a poorly designed template wrt exception specifications
template<class T>
bool operator==(const T& lhs, const T& rhs) throw()
{
  return &lhs == &rhs;
}
```

```
// Function pointer type for a window system callback
// when a window system event occurs
typedef void (*CallBackPtr)(int eventXLocation,
                            int eventYLocation,
                            void *dataToPassBack);

// Window system class for holding onto callback
// functions registered by window system clients
class CallBack {
public:
  CallBack(CallBackPtr fPtr, void *dataToPassBack)
  : func(fPtr), data(dataToPassBack) {}

  void makeCallBack(int eventXLocation,
                    int eventYLocation) const throw();

private:
  CallBackPtr func;               // function to call when
                                  // callback is made

  void *data;                     // data to pass to callback
};                                // function

// To implement the callback, we call the registered func-
// tion with event's coordinates and the registered data
void CallBack::makeCallBack(int eventXLocation,
                            int eventYLocation) const throw()
{
  func(eventXLocation, eventYLocation, data);
}
```

```
typedef void (*CallBackPtr)(int eventXLocation,
                            int eventYLocation,
                            void *dataToPassBack) throw();
```

```
// a callback function without an exception specification
void callBackFcn1(int eventXLocation, int eventYLocation,
                  void *dataToPassBack);

void *callBackData;

...

CallBack c1(callBackFcn1, callBackData);
                               // error! callBackFcn1
                               // might throw an exception

// a callback function with an exception specification
void callBackFcn2(int eventXLocation,
                  int eventYLocation,
                  void *dataToPassBack) throw();

CallBack c2(callBackFcn2, callBackData);
                               // okay, callBackFcn2 has a
                               // conforming ex. spec.
```

```
class UnexpectedException {};  // all unexpected exception
                               // objects will be replaced
                               // by objects of this type

void convertUnexpected()       // function to call if
{                              // an unexpected exception
  throw UnexpectedException(); // is thrown
}
```

```
void convertUnexpected()         // function to call if
{                                // an unexpected exception
  throw;                         // is thrown; just rethrow
}                                // the current exception

set_unexpected(convertUnexpected);
                                 // install convertUnexpected
                                 // as the unexpected
                                 // replacement
```

```
class Session {                        // for modeling online
public:                                // sessions
  ~Session();
  ...

private:
  static void logDestruction(Session *objAddr) throw();
};

Session::~Session()
{
  try {
    logDestruction(this);
  }
  catch (...) {}
}
```

```
class String { ... };    // a string class (the standard
                         // string type may be implemented
                         // as described below, but it
                         // doesn’t have to be)

String s1 = "Hello";

String s2 = s1;                 // call String copy ctor
```

```
cout << s1;                             // read s1’s value
cout << s1 + s2;                        // read s1’s and s2’s values
```

```
String s = "Homer’s Iliad";     // Assume s is a
                                // reference-counted string
...
cout << s[3];                   // call operator[] to read s[3]
s[3] = ’x’;                     // call operator[] to write s[3]
```

```
class LargeObject {              // large persistent objects
public:
  LargeObject(ObjectID id);      // restore object from disk

  const string& field1() const;  // value of field 1
  int field2() const;            // value of field 2
  double field3() const;         // ...
  const string& field4() const;
  const string& field5() const;
  ...

};
```

```
void restoreAndProcessObject(ObjectID id)
{
 LargeObject object(id);       // restore object
  ...
}
```

```
void restoreAndProcessObject(ObjectID id)
{
  LargeObject object(id);

  if (object.field2() == 0) {
    cout << "Object " << id << ": null field2.\n";
  }
}
```

```
class LargeObject {
public:
  LargeObject(ObjectID id);

  const string& field1() const;
  int field2() const;
  double field3() const;
  const string& field4() const;
  ...

private:
  ObjectID oid;

  mutable string *field1Value;    // see below for a
  mutable int *field2Value;       // discussion of "mutable"
  mutable double *field3Value;
  mutable string *field4Value;
  ...

};
LargeObject::LargeObject(ObjectID id)
: oid(id), field1Value(0), field2Value(0), field3Value(0), ...
{}

const string& LargeObject::field1() const
{
  if (field1Value == 0) {
    read the data for field 1 from the database and make
    field1Value point to it;
  }

  return *field1Value;
}
```

```
class LargeObject {
public:
  const string& field1() const;  // unchanged
  ...

private:
  string *field1Value;           // not declared mutable
  ...                            // so that older
};                               // compilers will accept it

const string& LargeObject::field1() const
{
  // declare a pointer, fakeThis, that points where this
  // does, but where the constness of the object has been
  // cast away
  LargeObject * const fakeThis =
    const_cast<LargeObject* const>(this);

  if (field1Value == 0) {
    fakeThis->field1Value =      // this assignment is OK,
      the appropriate data       // because what fakeThis
      from the database;         // points to isn't const
  }

  return *field1Value;
}
```

```
// Use of old-style cast to help emulate mutable
const string& LargeObject::field1() const
{
  LargeObject * const fakeThis = (LargeObject* const)this;
  ...                                  // as above
}
```

```
template<class T>
class Matrix { ... };              // for homogeneous matrices

Matrix<int> m1(1000, 1000);        // a 1000 by 1000 matrix
Matrix<int> m2(1000, 1000);        // ditto

...

Matrix<int> m3 = m1 + m2;          // add m1 and m2
```

```
Matrix<int> m4(1000, 1000);
...                                // give m4 some values
m3 = m4 * m1;
```

```
m3 = m1 + m2;                         // remember that m3 is the
                                      // sum of m1 and m2
m1 = m4;                              // now m3 is the sum of m2
                                      // and the OLD value of m1!
```

```
template<class NumericalType>
class DataCollection {
public:
  NumericalType min() const;
  NumericalType max() const;
  NumericalType avg() const;
  ...
};
```

```
int findCubicleNumber(const string& employeeName)
{
  // define a static map to hold (employee name, cubicle number)
  // pairs. This map is the local cache.
  typedef map<string, int> CubicleMap;
  static CubicleMap cubes;

  // try to find an entry for employeeName in the cache;
  // the STL iterator "it" will then point to the found
  // entry, if there is one (see Item 35 for details)
  CubicleMap::iterator it = cubes.find(employeeName);

  // "it"'s value will be cubes.end() if no entry was
  // found (this is standard STL behavior). If this is
  // the case, consult the database for the cubicle
  // number, then add it to the cache
  if (it == cubes.end()) {
    int cubicle =
      the result of looking up employeeName's cubicle
      number in the database;

    cubes[employeeName] = cubicle;    // add the pair
                                      // (employeeName, cubicle)
                                      // to the cache
    return cubicle;
  }
  else {
    // "it" points to the correct cache entry, which is a
    // (employee name, cubicle number) pair. We want only
    // the second component of this pair, and the member
    // "second" will give it to us
    return (*it).second;
  }
}
```

```
template<class T>                // template for dynamic
class DynArray { ... };          // array-of-T classes

DynArray<double> a;              // at this point, only a[0]
                                 // is a legitimate array
                                 // element

a[22] = 3.5;                     // a is automatically
                                 // extended: valid indices
                                 // are now 0-22

a[32] = 0;                       // a extends itself again;
                                 // now a[0]-a[32] are valid
```

```
template<class T>
T& DynArray<T>::operator[](int index)
{
  if (index < 0) {
    throw an exception;           // negative indices are
  }                               // still invalid

  if (index > the current maximum index value) {
    call new to allocate enough additional memory so that
    index is valid;
  }

  return the indexth element of the array;
}
```

```
template<class T>
T& DynArray<T>::operator[](int index)
{
  if (index < 0) throw an exception;

  if (index > the current maximum index value) {
    int diff = index - the current maximum index value;

    call new to allocate enough additional memory so that
    index+diff is valid;
  }

  return the indexth element of the array;
}
```

```
DynArray<double> a;               // only a[0] is valid

a[22] = 3.5;                      // new is called to expand
                                  // a’s storage through
                                  // index 44; a’s logical
                                  // size becomes 23

a[32] = 0;                        // a’s logical size is
                                  // changed to allow a[32],
                                  // but new isn’t called
// returns the number of occurrences of ch in str
size_t countChar(const string& str, char ch);

char buffer[MAX_STRING_LEN];
char c;

// read in a char and a string; use setw to avoid
// overflowing buffer when reading the string
cin >> c >> setw(MAX_STRING_LEN) >> buffer;

cout << "There are " << countChar(buffer, c)
     << " occurrences of the character " << c
     << " in " << buffer << endl;
```

```
void uppercasify(string& str);     // changes all chars in
                                   // str to upper case
```

```
char subtleBookPlug[] = "Effective C++";
uppercasify(subtleBookPlug);       // error!
```

```
const Number operator+(const Number& lhs,
                       const Number& rhs);
```

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  ...
  int numerator() const;
  int denominator() const;
};

// For an explanation of why the return value is const,
// see Item 6
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
```

```
// an unreasonable way to avoid returning an object
const Rational * operator*(const Rational& lhs,
                           const Rational& rhs);

Rational a = 10;
Rational b(1, 2);

Rational c = *(a * b);             // Does this look "natural"
                                   // to you?
// a dangerous (and incorrect) way to avoid returning
// an object
const Rational& operator*(const Rational& lhs,
                          const Rational& rhs);

Rational a = 10;
Rational b(1, 2);

Rational c = a * b;            // looks perfectly reasonable
```

```
// another dangerous (and incorrect) way to avoid
// returning an object
const Rational& operator*(const Rational& lhs,
                          const Rational& rhs)
{
 Rational result(lhs.numerator() * rhs.numerator(),
                 lhs.denominator() * rhs.denominator());
 return result;
}
```

```
// an efficient and correct way to implement a
// function that returns an object
const Rational operator*(const Rational& lhs,
                         const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}
```

```
Rational(lhs.numerator() * rhs.numerator(),
         lhs.denominator() * rhs.denominator());
```

```
Rational a = 10;
Rational b(1, 2);
Rational c = a * b;              // operator* is called here
```

```
// the most efficient way to write a function returning
// an object
inline const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}
```

```
class UPInt {                          // class for unlimited
public:                                // precision integers
  UPInt();
  UPInt(int value);
  ...

};

// For an explanation of why the return value is const,
// see Item 6
const UPInt operator+(const UPInt& lhs, const UPInt& rhs);

UPInt upi1, upi2;

...

UPInt upi3 = upi1 + upi2;
```

```
const UPInt operator+(const UPInt& lhs,      // add UPInt
                      const UPInt& rhs);     // and UPInt

const UPInt operator+(const UPInt& lhs,      // add UPInt
                      int rhs);              // and int

const UPInt operator+(int lhs,               // add int and
                      const UPInt& rhs);     // UPInt

UPInt upi1, upi2;

...

UPInt upi3 = upi1 + upi2;       // fine, no temporary for
                                // upi1 or upi2

upi3 = upi1 + 10;               // fine, no temporary for
                                // upi1 or 10

upi3 = 10 + upi2;               // fine, no temporary for
                                // 10 or upi2
```

```
class Rational {
public:
  ...
  Rational& operator+=(const Rational& rhs);
  Rational& operator-=(const Rational& rhs);
};

// operator+ implemented in terms of operator+=; see
// Item 6 for an explanation of why the return value is
// const and page 109B for a warning about implementation
const Rational operator+(const Rational& lhs,
                         const Rational& rhs)
{
  return Rational(lhs) += rhs;
}

// operator- implemented in terms of operator -=
const Rational operator-(const Rational& lhs,
                         const Rational& rhs)
{
  return Rational(lhs) -= rhs;
}
```

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{
 return T(lhs) += rhs;          // see discussion below
}

template<class T>
const T operator-(const T& lhs, const T& rhs)
{
 return T(lhs) -= rhs;          // see discussion below
}

...
```

```
Rational a, b, c, d, result;
...
result = a + b + c + d;       // probably uses 3 temporary
                              // objects, one for each call
                              // to operator+
result = a;                           // no temporary needed
result += b;                          // no temporary needed
result += c;                          // no temporary needed
result += d;                          // no temporary needed
```

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{ return T(lhs) += rhs; }
```

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{
  T result(lhs);               // copy lhs into result
  return result += rhs;        // add rhs to it and return
}
```

```
#ifdef STDIO
#include <stdio.h>
#else
#include <iostream>
#include <iomanip>
using namespace std;
#endif

const int VALUES = 30000;       // # of values to read/write

int main()
{
  double d;

  for (int n = 1; n <= VALUES; ++n) {
#ifdef STDIO
    scanf("%lf", &d);
    printf("%10.5f", d);
#else
    cin >> d;
    cout << setw(10)                      // set field width
         << setprecision(5)               // set decimal places
         << setiosflags(ios::showpoint)   // keep trailing 0s
         << setiosflags(ios::fixed)       // use these settings
         << d;
#endif

    if (n % 5 == 0) {
#ifdef STDIO
      printf("\n");
#else
      cout << '\n';
#endif
    }
  }

  return 0;
}
```

```
0.00000    0.69315    1.09861    1.38629    1.60944
1.79176    1.94591    2.07944    2.19722    2.30259
2.39790    2.48491    2.56495    2.63906    2.70805
2.77259    2.83321    2.89037    2.94444    2.99573
3.04452    3.09104    3.13549    3.17805    3.21888
```

```
cout << setw(10)
     << setprecision(5)
     << setiosflags(ios::showpoint)
     << setiosflags(ios::fixed)
     << d;
```

```
class C1 {
public:
  C1();

  virtual ~C1();
  virtual void f1();
  virtual int f2(char c) const;
  virtual void f3(const string& s);

  void f4() const;

  ...
};
```

```
class C2: public C1 {
public:
  C2();                           // nonvirtual function
  virtual ~C2();                  // redefined function
  virtual void f1();              // redefined function
  virtual void f5(char *str);   // new virtual function
  ...
};
```

```
(*pC1->vptr[i])(pC1);  // call the function pointed to by the
                       // i-th entry in the vtbl pointed to
                       // by pC1->vptr; pC1 is passed to the
                       // function as the "this" pointer
```

```
class NLComponent {               // abstract base class for
public:                           // newsletter components

  ...                             // contains at least one
};                                // pure virtual function

class TextBlock: public NLComponent {
public:
  ...                             // contains no pure virtual
};                                // functions

class Graphic: public NLComponent {
public:
  ...                             // contains no pure virtual
};                                // functions

class NewsLetter {                // a newsletter object
public:                           // consists of a list of
  ...                             // NLComponent objects

private:
  list<NLComponent*> components;
};
```

```
NewsLetter::NewsLetter(istream& str)
{
  while (str) {
    read the next component object from str;
    add the object to the list of this
    newsletter’s components;
  }
}
```

```
class NewsLetter {
public:
  ...

private:
  // read the data for the next NLComponent from str,
  // create the component and return a pointer to it
  static NLComponent * readComponent(istream& str);
  ...
};

NewsLetter::NewsLetter(istream& str)
{
  while (str) {
    // add the pointer returned by readComponent to the
    // end of the components list; "push_back" is a list
    // member function that inserts at the end of the list
    components.push_back(readComponent(str));
  }
}
```

```
class NLComponent {
public:
  // declaration of virtual copy constructor
  virtual NLComponent * clone() const = 0;
  ...

};

class TextBlock: public NLComponent {
public:
  virtual TextBlock * clone() const     // virtual copy
  { return new TextBlock(*this); }      // constructor
  ...

};

class Graphic: public NLComponent {
public:
  virtual Graphic * clone() const       // virtual copy
  { return new Graphic(*this); }        // constructor
  ...

};
```

```
class NewsLetter {
public:
  NewsLetter(const NewsLetter& rhs);
  ...

private:
  list<NLComponent*> components;
};

NewsLetter::NewsLetter(const NewsLetter& rhs)
{
  // iterate over rhs's list, using each element's
  // virtual copy constructor to copy the element into
  // the components list for this object. For details on
  // how the following code works, see Item 35.
  for (list<NLComponent*>::const_iterator it =
         rhs.components.begin();
       it != rhs.components.end();
       ++it) {

    // "it" points to the current element of rhs.components,
    // so call that element's clone function to get a copy
    // of the element, and add that copy to the end of
    // this object's list of components
    components.push_back((*it)->clone());
  }
}
```

```
class NLComponent {
public:
  // unconventional declaration of output operator
  virtual ostream& operator<<(ostream& str) const = 0;
  ...
};

class TextBlock: public NLComponent {
public:
  // virtual output operator (also unconventional)
  virtual ostream& operator<<(ostream& str) const;
};

class Graphic: public NLComponent {
public:
  // virtual output operator (still unconventional)
  virtual ostream& operator<<(ostream& str) const;
};

TextBlock t;
Graphic g;

...

t << cout;                      // print t on cout via
                                // virtual operator<<; note
                                // unconventional syntax

g << cout;                      // print g on cout via
                                // virtual operator<<; note
                                // unconventional syntax
```

```
class NLComponent {
public:
  virtual ostream& print(ostream& s) const = 0;
  ...

};

class TextBlock: public NLComponent {
public:
  virtual ostream& print(ostream& s) const;
  ...

};

class Graphic: public NLComponent {
public:
  virtual ostream& print(ostream& s) const;
  ...

};

inline
ostream& operator<<(ostream& s, const NLComponent& c)
{
  return c.print(s);
}
```

```
class CantBeInstantiated {
private:
  CantBeInstantiated();
  CantBeInstantiated(const CantBeInstantiated&);

  ...

};
```

```
class PrintJob;                    // forward declaration

class Printer {
public:
  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();

  ...

friend Printer& thePrinter();

private:
  Printer();
  Printer(const Printer& rhs);

  ...

};

Printer& thePrinter()
{
  static Printer p;               // the single printer object
  return p;
}
```

```
class PrintJob {
public:
  PrintJob(const string& whatToPrint);
  ...

};

string buffer;

...                                 // put stuff in buffer

thePrinter().reset();
thePrinter().submitJob(buffer);
```

```
class Printer {
public:
  static Printer& thePrinter();
  ...

private:
  Printer();
  Printer(const Printer& rhs);
  ...

};

Printer& Printer::thePrinter()
{
  static Printer p;
  return p;
}
```

```
Printer::thePrinter().reset();
Printer::thePrinter().submitJob(buffer);
```

```
namespace PrintingStuff {
  class Printer {                // this class is in the
  public:                        // PrintingStuff namespace

    void submitJob(const PrintJob& job);
    void reset();
    void performSelfTest();
    ...

  friend Printer& thePrinter();
  private:
    Printer();
    Printer(const Printer& rhs);
    ...

  };

  Printer& thePrinter()          // so is this function
  {
    static Printer p;
    return p;
  }
}                                // this is the end of the
                                 // namespace
```

```
PrintingStuff::thePrinter().reset();
PrintingStuff::thePrinter().submitJob(buffer);
```

```
using PrintingStuff::thePrinter; // import the name
                                 // "thePrinter" from the
                                 // namespace "PrintingStuff"
                                 // into the current scope

thePrinter().reset();            // now thePrinter can be
thePrinter().submitJob(buffer);  // used as if it were a
                                 // local name
```

```
class Printer {
public:
  class TooManyObjects{};        // exception class for use
                                 // when too many objects
                                 // are requested
  Printer();
  ~Printer();

  ...

private:
  static size_t numObjects;

  Printer(const Printer& rhs);  // there is a limit of 1
                                 // printer, so never allow
};                               // copying
```

```
// Obligatory definition of the class static
size_t Printer::numObjects = 0;

Printer::Printer()
{
  if (numObjects >= 1) {
    throw TooManyObjects();
  }

  proceed with normal construction here;

  ++numObjects;
}

Printer::~Printer()
{
  perform normal destruction here;

  --numObjects;
}
```

```
class CPFMachine {          // for machines that can
private:                    // copy, print, and fax

  Printer p;                // for printing capabilities
  FaxMachine f;             // for faxing capabilities
  CopyMachine c;            // for copying capabilities

  ...

};

CPFMachine m1;              // fine

CPFMachine m2;              // throws TooManyObjects exception
```

```
class FSA {
public:
  // pseudo-constructors
  static FSA * makeFSA();
  static FSA * makeFSA(const FSA& rhs);
  ...

private:
  FSA();
  FSA(const FSA& rhs);
  ...
};

FSA * FSA::makeFSA()
{ return new FSA(); }

FSA * FSA::makeFSA(const FSA& rhs)
{ return new FSA(rhs); }
```

```
// indirectly call default FSA constructor
auto_ptr<FSA> pfsa1(FSA::makeFSA());

// indirectly call FSA copy constructor
auto_ptr<FSA> pfsa2(FSA::makeFSA(*pfsa1));

...                // use pfsa1 and pfsa2 as normal pointers,
                   // but don't worry about deleting them
```

```
class Printer {
public:
  class TooManyObjects{};

  // pseudo-constructor
  static Printer * makePrinter();

  ~Printer();

  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
  ...

private:
  static size_t numObjects;

  Printer();

  Printer(const Printer& rhs); // we don't define this
};                             // function, because we'll
                               // never allow copying

// Obligatory definition of class static
size_t Printer::numObjects = 0;

Printer::Printer()
{
  if (numObjects >= 1) {
    throw TooManyObjects();
  }

  proceed with normal object construction here;

  ++numObjects;
}

Printer * Printer::makePrinter()
{ return new Printer; }
```

```
Printer p1;                       // error! default ctor is
                                  // private

Printer *p2 =
  Printer::makePrinter();         // fine, indirectly calls
                                  // default ctor

Printer p3 = *p2;                 // error! copy ctor is
                                  // private

p2->performSelfTest();            // all other functions are
p2->reset();                      // called as usual

...

delete p2;                        // avoid resource leak; this
                                  // would be unnecessary if
                                  // p2 were an auto_ptr
```

```
class Printer {
public:
  class TooManyObjects{};

  // pseudo-constructors
  static Printer * makePrinter();
  static Printer * makePrinter(const Printer& rhs);

  ...

private:
  static size_t numObjects;
  static const size_t maxObjects = 10;    // see below

  Printer();
  Printer(const Printer& rhs);
};

// Obligatory definitions of class statics
size_t Printer::numObjects = 0;
const size_t Printer::maxObjects;

Printer::Printer()
{
  if (numObjects >= maxObjects) {
    throw TooManyObjects();
  }

  ...

}

Printer::Printer(const Printer& rhs)
```

```
{
  if (numObjects >= maxObjects) {
    throw TooManyObjects();
  }

  ...

}

Printer * Printer::makePrinter()
{ return new Printer; }

Printer * Printer::makePrinter(const Printer& rhs)
{ return new Printer(rhs); }
```

```
class Printer {
private:
  enum { maxObjects = 10 };      // within this class,
  ...                            // maxObjects is the
};                               // constant 10
```

```
class Printer {
private:
  static const size_t maxObjects;    // no initial value given

  ...

};

// this goes in a single implementation file
const size_t Printer::maxObjects = 10;
```

```
template<class BeingCounted>
class Counted {
public:
  class TooManyObjects{};        // for throwing exceptions

  static size_t objectCount() { return numObjects; }

protected:
  Counted();
  Counted(const Counted& rhs);

  ~Counted() { --numObjects; }

private:
  static size_t numObjects;
  static const size_t maxObjects;

  void init();                   // to avoid ctor code
};                               // duplication

template<class BeingCounted>
Counted<BeingCounted>::Counted()
{ init(); }

template<class BeingCounted>
Counted<BeingCounted>::Counted(const Counted<BeingCounted>&)
{ init(); }

template<class BeingCounted>
void Counted<BeingCounted>::init()
{
  if (numObjects >= maxObjects) throw TooManyObjects();
  ++numObjects;
}
```

```
class Printer: private Counted<Printer> {
public:
  // pseudo-constructors
  static Printer * makePrinter();
  static Printer * makePrinter(const Printer& rhs);

  ~Printer();

  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
  ...

  using Counted<Printer>::objectCount;      // see below
  using Counted<Printer>::TooManyObjects;   // see below

private:
  Printer();
  Printer(const Printer& rhs);
};
```

```
class Printer: private Counted<Printer> {
public:
  ...
  using Counted<Printer>::objectCount;  // make this function
                                        // public for clients
  ...                                   // of Printer
};
```

```
class Printer: private Counted<Printer> {
public:
  ...
  Counted<Printer>::objectCount;       // make objectCount
                                       // public in Printer
  ...

};
```

```
template<class BeingCounted>                // defines numObjects
size_t Counted<BeingCounted>::numObjects;  // and automatically
                                            // initializes it to 0
```

```
class UPNumber {
public:
  UPNumber();
  UPNumber(int initValue);
  UPNumber(double initValue);
  UPNumber(const UPNumber& rhs);

  // pseudo-destructor (a const member function, because
  // even const objects may be destroyed)
  void destroy() const { delete this; }

  ...

private:
  ~UPNumber();
};
```

```
UPNumber n;                       // error! (legal here, but
                                  // illegal when n's dtor is
                                  // later implicitly invoked)

UPNumber *p = new UPNumber;       // fine

...

delete p;                         // error! attempt to call
                                  // private destructor

p->destroy();                     // fine
```

```
class UPNumber { ... };           // declares dtor or ctors
                                  // private

class NonNegativeUPNumber:
  public UPNumber { ... };        // error! dtor or ctors
                                  // won't compile
class Asset {
private:
  UPNumber value;
  ...                             // error! dtor or ctors
                                  // won't compile
};
```

```
class UPNumber { ... };          // declares dtor protected

class NonNegativeUPNumber:
  public UPNumber { ... };       // now okay; derived
                                 // classes have access to
                                 // protected members

class Asset {
public:
  Asset(int initValue);
  ~Asset();
  ...

private:
  UPNumber *value;
};

Asset::Asset(int initValue)
: value(new UPNumber(initValue))    // fine
{ ... }

Asset::~Asset()
{ value->destroy(); }               // also fine
```

```
NonNegativeUPNumber *n1 =
  new NonNegativeUPNumber;          // on heap
NonNegativeUPNumber n2;             // not on heap
```

```
class UPNumber {
public:
  // exception to throw if a non-heap object is created
  class HeapConstraintViolation {};

  static void * operator new(size_t size);

  UPNumber();
  ...

private:
  static bool onTheHeap;          // inside ctors, whether
                                  // the object being
  ...                             // constructed is on heap

};

// obligatory definition of class static
bool UPNumber::onTheHeap = false;

void *UPNumber::operator new(size_t size)
{
  onTheHeap = true;
  return ::operator new(size);
}

UPNumber::UPNumber()
{
  if (!onTheHeap) {
    throw HeapConstraintViolation();
  }

  proceed with normal construction here;

  onTheHeap = false;            // clear flag for next obj.
}
```

```
// incorrect attempt to determine whether an address
// is on the heap
bool onHeap(const void *address)
{
  char onTheStack;                 // local stack variable

  return address < &onTheStack;
}
```

```
void allocateSomeObjects()
{
  char *pc = new char;          // heap object: onHeap(pc)
                                // will return true

  char c;                       // stack object: onHeap(&c)
                                // will return false

  static char sc;               // static object: onHeap(&sc)
                                // will return true

  ...
}
```

```
void *operator new(size_t size)
{
  void *p = getMemory(size);       // call some function to
                                   // allocate memory and
                                   // handle out-of-memory
                                   // conditions

  add p to the collection of allocated addresses;

  return p;

}

void operator delete(void *ptr)
{
  releaseMemory(ptr);              // return memory to
                                   // free store

  remove ptr from the collection of allocated addresses;
}

bool isSafeToDelete(const void *address)
{
  return whether address is in collection of
  allocated addresses;
}
```

```
class HeapTracked {            // mixin class; keeps track of
public:                        // ptrs returned from op. new

  class MissingAddress{};      // exception class; see below

  virtual ~HeapTracked() = 0;

  static void *operator new(size_t size);
  static void operator delete(void *ptr);

  bool isOnHeap() const;

private:
  typedef const void* RawAddress;
  static list<RawAddress> addresses;
};
```

```
// mandatory definition of static class member
list<RawAddress> HeapTracked::addresses;
// HeapTracked's destructor is pure virtual to make the
// class abstract. The destructor must still be
// defined, however, so we provide this empty definition.
HeapTracked::~HeapTracked() {}

void * HeapTracked::operator new(size_t size)
{
  void *memPtr = ::operator new(size);// get the memory
  addresses.push_front(memPtr);         // put its address at
                                        // the front of the list
  return memPtr;
}
void HeapTracked::operator delete(void *ptr)
{
  // gracefully hande null pointers
  if (ptr == 0) return;
  // get an "iterator" that identifies the list
  // entry containing ptr; see Item 35 for details
  list<RawAddress>::iterator it =
    find(addresses.begin(), addresses.end(), ptr);
   if (it != addresses.end()) {    // if an entry was found
    addresses.erase(it);           // remove the entry
    ::operator delete(ptr);        // deallocate the memory
  } else {                         // otherwise
```

```
    throw MissingAddress();           // ptr wasn't allocated by
  }                                   // op. new, so throw an
}                                     // exception

bool HeapTracked::isOnHeap() const
{
  // get a pointer to the beginning of the memory
  // occupied by *this; see below for details
  const void *rawAddress = dynamic_cast<const void*>(this);

  // look up the pointer in the list of addresses
  // returned by operator new
  list<RawAddress>::iterator it =
    find(addresses.begin(), addresses.end(), rawAddress);

  return it != addresses.end();       // return whether it was
}                                     // found
```

```
class Asset: public HeapTracked {
private:
  UPNumber value;
  ...
};
```

```
void inventoryAsset(const Asset *ap)
{
  if (ap->isOnHeap()) {
    ap is a heap-based asset — inventory it as such;
  }
  else {
    ap is a non-heap-based asset — record it that way;
  }
}
```

```
class UPNumber {
private:
  static void *operator new(size_t size);
  static void operator delete(void *ptr);
  ...
};
```

```
UPNumber n1;                  // okay
static UPNumber n2;           // also okay
UPNumber *p = new UPNumber;   // error! attempt to call
                              // private operator new
```

```
class UPNumber { ... };          // as above

class NonNegativeUPNumber:       // assume this class
  public UPNumber {              // declares no operator new
  ...
};

NonNegativeUPNumber n1;          // okay

static NonNegativeUPNumber n2; // also okay

NonNegativeUPNumber *p =         // error! attempt to call
  new NonNegativeUPNumber;       // private operator new
```

```
class Asset {
public:
  Asset(int initValue);
  ...

private:
  UPNumber value;
};

Asset *pa = new Asset(100);     // fine, calls
                                // Asset::operator new or
                                // ::operator new, not
                                // UPNumber::operator new
```

```
template<class T>                // template for smart
class SmartPtr {                 // pointer objects
public:
  SmartPtr(T* realPtr = 0);      // create a smart ptr to an
                                 // obj given a dumb ptr to
                                 // it; uninitialized ptrs
                                 // default to 0 (null)

  SmartPtr(const SmartPtr& rhs); // copy a smart ptr

  ~SmartPtr();                   // destroy a smart ptr

  // make an assignment to a smart ptr
  SmartPtr& operator=(const SmartPtr& rhs);

  T* operator->() const;         // dereference a smart ptr
                                 // to get at a member of
                                 // what it points to

  T& operator*() const;          // dereference a smart ptr

private:
  T *pointee;                    // what the smart ptr
};                               // points to
```

```
template<class T>                    // template for smart ptrs
class DBPtr {                        // to objects in a
public:                              // distributed DB

  DBPtr(T *realPtr = 0);             // create a smart ptr to a
                                     // DB object given a local
                                     // dumb pointer to it

  DBPtr(DataBaseID id);              // create a smart ptr to a
                                     // DB object given its
                                     // unique DB identifier

  ...                                // other smart ptr
};                                   // functions as above

class Tuple {                        // class for database
public:                              // tuples
  ...
  void displayEditDialog();          // present a graphical
                                     // dialog box allowing a
                                     // user to edit the tuple

  bool isValid() const;              // return whether *this
};                                   // passes validity check

// class template for making log entries whenever a T
// object is modified; see below for details
template<class T>
class LogEntry {
public:
  LogEntry(const T& objectToBeModified);
  ~LogEntry();
};

void editTuple(DBPtr<Tuple>& pt)
{
  LogEntry<Tuple> entry(*pt);   // make log entry for this
                                // editing operation; see
                                // below for details

  // repeatedly display edit dialog until valid values
  // are provided
  do {
    pt->displayEditDialog();
  } while (pt->isValid() == false);
}
```

```
template<class T>
class auto_ptr {
public:
  auto_ptr(T *ptr = 0): pointee(ptr) {}
  ~auto_ptr() { delete pointee; }
  ...

private:
  T *pointee;
};
```

```
auto_ptr<TreeNode> ptn1(new TreeNode);

auto_ptr<TreeNode> ptn2 = ptn1;    // call to copy ctor;
                                   // what should happen?

auto_ptr<TreeNode> ptn3;

ptn3 = ptn2;                       // call to operator=;
                                   // what should happen?
```

```
template<class T>
class auto_ptr {
public:
  ...

  auto_ptr(auto_ptr<T>& rhs);       // copy constructor

  auto_ptr<T>&                      // assignment
  operator=(auto_ptr<T>& rhs);      // operator

  ...
};

template<class T>
auto_ptr<T>::auto_ptr(auto_ptr<T>& rhs)
{
  pointee = rhs.pointee;       // transfer ownership of
                               // *pointee to *this

  rhs.pointee = 0;             // rhs no longer owns
}                              // anything

template<class T>
auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs)
{
  if (this == &rhs)            // do nothing if this
    return *this;              // object is being assigned
                               // to itself

  delete pointee;              // delete currently owned
                               // object

  pointee = rhs.pointee;       // transfer ownership of
  rhs.pointee = 0;             // *pointee from rhs to *this

  return *this;
}
```

```
// this function will often lead to disaster
void printTreeNode(ostream& s, auto_ptr<TreeNode> p)
{ s << *p; }

int main()
{
  auto_ptr<TreeNode> ptn(new TreeNode);

  ...

  printTreeNode(cout, ptn);      // pass auto_ptr by value

  ...

}
```

```
// this function behaves much more intuitively
void printTreeNode(ostream& s,
                   const auto_ptr<TreeNode>& p)
{ s << *p; }
```

```
template<class T>
SmartPtr<T>::~SmartPtr()
{
  if (*this owns *pointee) {
    delete pointee;
  }
}
```

```
template<class T>
T& SmartPtr<T>::operator*() const
{
  perform "smart pointer" processing;

  return *pointee;
}
```

```
void editTuple(DBPtr<Tuple>& pt)
{
  LogEntry<Tuple> entry(*pt);

  do {
    pt->displayEditDialog();
  } while (pt->isValid() == false);
}
```

```
template<class T>
T* SmartPtr<T>::operator->() const
{
  perform "smart pointer" processing;

  return pointee;
}
```

```
SmartPtr<TreeNode> ptn;
...
if (ptn == 0) ...                  // error!
if (ptn) ...                       // error!
if (!ptn) ...                      // error!
```

```
template<class T>
class SmartPtr {
public:
  ...
  operator void*();                   // returns 0 if the smart
  ...                                 // ptr is null, nonzero
};                                    // otherwise

SmartPtr<TreeNode> ptn;

...

if (ptn == 0) ...                     // now fine

if (ptn) ...                          // also fine

if (!ptn) ...                         // fine
```

```
ifstream inputFile("datafile.dat");
if (inputFile) ...                // test to see if inputFile
                                  // was successfully
                                  // opened
```

```
SmartPtr<Apple> pa;
SmartPtr<Orange> po;

...

if (pa == po) ...                // this compiles!
```

```
template<class T>
class SmartPtr {
public:
  ...
  bool operator!() const;        // returns true if and only
  ...                            // if the smart ptr is null

};
```

```
SmartPtr<TreeNode> ptn;

...

if (!ptn) {                          // fine
  ...                                // ptn is null
}
else {
  ...                                // ptn is not null
}
```

```
if (ptn == 0) ...                    // still an error
if (ptn) ...                         // also an error
```

```
SmartPtr<Apple> pa;
SmartPtr<Orange> po;

...

if (!pa == !po) ...                 // alas, this compiles
```

```
class Tuple { ... };                // as before

void normalize(Tuple *pt);          // put *pt into canonical
                                    // form; note use of dumb
                                    // pointer
```

```
DBPtr<Tuple> pt;
...
normalize(pt);                        // error!
```

```
template<class T>                  // as before
class DBPtr {
public:
  ...
  operator T*() { return pointee; }
  ...
};

DBPtr<Tuple> pt;

...

normalize(pt);                     // this now works
```

```
if (pt == 0) ...                  // fine, converts pt to a
                                  // Tuple*
if (pt) ...                       // ditto
if (!pt) ...                      // ditto (reprise)
```

```
void processTuple(DBPtr<Tuple>& pt)
{
 Tuple *rawTuplePtr = pt;      // converts DBPtr<Tuple> to
                               // Tuple*

 use rawTuplePtr to modify the tuple;

}
```

```
class TupleAccessors {
public:
  TupleAccessors(const Tuple *pt);   // pt identifies the
  ...                                // tuple whose accessors
};                                   // we care about
```

```
TupleAccessors merge( const TupleAccessors& ta1,
                      const TupleAccessors& ta2);
```

```
Tuple *pt1, *pt2;

...

merge(pt1, pt2);          // fine, both pointers are converted
                          // to TupleAccessors objects
```

```
DBPtr<Tuple> pt1, pt2;
...
merge(pt1, pt2);          // error! No way to convert pt1 and
                          // pt2 to TupleAccessors objects
```

```
DBPtr<Tuple> pt = new Tuple;
...
delete pt;
```

```
class MusicProduct {
public:
  MusicProduct(const string& title);
  virtual void play() const = 0;
  virtual void displayTitle() const = 0;
  ...
};

class Cassette: public MusicProduct {
public:
  Cassette(const string& title);
  virtual void play() const;
  virtual void displayTitle() const;
  ...
};

class CD: public MusicProduct {
public:
  CD(const string& title);
  virtual void play() const;
  virtual void displayTitle() const;
  ...
};
```

```
void displayAndPlay(const MusicProduct* pmp, int numTimes)
{
  for (int i = 1; i <= numTimes; ++i) {
    pmp->displayTitle();
    pmp->play();
  }
}
```

```
Cassette *funMusic = new Cassette("Alapalooza");
CD *nightmareMusic = new CD("Disco Hits of the 70s");

displayAndPlay(funMusic, 10);
displayAndPlay(nightmareMusic, 0);
```

```
void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int numTimes);

SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));
SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the 70s"));

displayAndPlay(funMusic, 10);         // error!
displayAndPlay(nightmareMusic, 0);    // error!
```

```
class SmartPtr<Cassette> {
public:
  operator SmartPtr<MusicProduct>()
  { return SmartPtr<MusicProduct>(pointee); }

  ...

private:
  Cassette *pointee;
};

class SmartPtr<CD> {
public:
  operator SmartPtr<MusicProduct>()
  { return SmartPtr<MusicProduct>(pointee); }

  ...

private:
  CD *pointee;
};
```

```
template<class T>                    // template class for smart
class SmartPtr {                     // pointers-to-T objects
public:
  SmartPtr(T* realPtr = 0);

  T* operator->() const;
  T& operator*() const;

  template<class newType>            // template function for
  operator SmartPtr<newType>()       // implicit conversion ops.
  {
    return SmartPtr<newType>(pointee);
  }

  ...
};
```

```
void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int howMany);

SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));
SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the 70s"));

displayAndPlay(funMusic, 10);        // used to be an error
displayAndPlay(nightmareMusic, 0);   // used to be an error
```

```
SmartPtr<Cassette>::operator SmartPtr<MusicProduct>()
{
  return SmartPtr<MusicProduct>(pointee);
}
```

```
template<class T>           // as above, including member tem-
class SmartPtr { ... };    // plate for conversion operators

void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int howMany);

void displayAndPlay(const SmartPtr<Cassette>& pc,
                    int howMany);

SmartPtr<CasSingle> dumbMusic(new CasSingle("Achy Breaky Heart"));

displayAndPlay(dumbMusic, 1);   // error!
```

```
CD goodCD("Flood");

const CD *p;                     // p is a non-const pointer
                                 // to a const CD object

CD * const p = &goodCD;          // p is a const pointer to
                                 // a non-const CD object;
                                 // because p is const, it
                                 // must be initialized

const CD * const p = &goodCD;    // p is a const pointer to
                                 // a const CD object
```

```
const SmartPtr<CD> p =           // p is a const smart ptr
  &goodCD;                       // to a non-const CD object
```

```
SmartPtr<const CD> p =              // p is a non-const smart ptr
  &goodCD;                          // to a const CD object
```

```
SmartPtr<CD> p;                          // non-const object,
                                         // non-const pointer

SmartPtr<const CD> p;                    // const object,
                                         // non-const pointer

const SmartPtr<CD> p = &goodCD;          // non-const object,
                                         // const pointer

const SmartPtr<const CD> p = &goodCD; // const object,
                                         // const pointer
```

```
CD *pCD = new CD("Famous Movie Themes");
const CD * pConstCD = pCD;              // fine
```

```
SmartPtr<CD> pCD = new CD("Famous Movie Themes");
SmartPtr<const CD> pConstCD = pCD;    // fine?
```

```
template<class T>                    // smart pointers to const
class SmartPtrToConst {              // objects

  ...                                // the usual smart pointer
                                     // member functions

protected:
  union {
    const T* constPointee;           // for SmartPtrToConst access
    T* pointee;                      // for SmartPtr access
  };
};

template<class T>                    // smart pointers to
class SmartPtr:                      // non-const objects
  public SmartPtrToConst<T> {
  ...                                // no data members
};
```

```
SmartPtr<CD> pCD = new CD("Famous Movie Themes");
SmartPtrToConst<CD> pConstCD = pCD;   // fine
```

```
class String {          // the standard string type may
public:                 // employ the techniques in this
                        // Item, but that is not required

  String(const char *value = "");
  String& operator=(const String& rhs);
  ...

private:
  char *data;
};

String a, b, c, d, e;

a = b = c = d = e = "Hello";
```

```
String& String::operator=(const String& rhs)
{
  if (this == &rhs) return *this;

  delete [] data;
  data = new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);

  return *this;
}
```

```
class String {
public:
  ...                               // the usual String member
                                    // functions go here

private:
  struct StringValue { ... };       // holds a reference count
                                    // and a string value

  StringValue *value;               // value of this String
};
```

```
class String {
private:
  struct StringValue {
    size_t refCount;
    char *data;

    StringValue(const char *initValue);
    ~StringValue();

  };

  ...

};

String::StringValue::StringValue(const char *initValue)
: refCount(1)
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}

String::StringValue::~StringValue()
{
  delete [] data;
}
```

```
class String {
public:
  String(const char *initValue = "");
  String(const String& rhs);

  ...

};
```

```
String::String(const String& rhs)
: value(rhs.value)
{
  ++value->refCount;
}
```

```
class String {
public:
  ~String();
  ...
};

String::~String()
{
  if (--value->refCount == 0) delete value;
}
```

```
class String {
public:
  String& operator=(const String& rhs);
  ...
};
```

```
String& String::operator=(const String& rhs)
{
  if (value == rhs.value) {      // do nothing if the values
    return *this;                // are already the same; this
  }                              // subsumes the usual test of
                                 // this against &rhs

  if (--value->refCount == 0) {  // destroy *this's value if
    delete value;                // no one else is using it
  }

  value = rhs.value;             // have *this share rhs's
  ++value->refCount;             // value

  return *this;
}
```

```
class String {
public:
  const char&
    operator[](int index) const;    // for const Strings

  char& operator[](int index);      // for non-const Strings

  ...

};
```

```
const char& String::operator[](int index) const
{
  return value->data[index];
}
```

```
String s;

...

cout << s[3];                              // this is a read
s[5] = 'x';                                // this is a write
```

```
char& String::operator[](int index)
{
  // if we’re sharing a value with other String objects,
  // break off a separate copy of the value for ourselves
  if (value->refCount > 1) {
    --value->refCount;        // decrement current value’s
                              // refCount, because we won’t
                              // be using that value any more

    value =                             // make a copy of the
      new StringValue(value->data);     // value for ourselves
  }

  // return a reference to a character inside our
  // unshared StringValue object
  return value->data[index];
}
```

```
class String {
private:
  struct StringValue {
    size_t refCount;
    bool shareable;                         // add this
    char *data;

    StringValue(const char *initValue);
    ~StringValue();

  };

...

};

String::StringValue::StringValue(const char *initValue)
:   refCount(1),
    shareable(true)                         // add this
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}

String::StringValue::~StringValue()
{
  delete [] data;
}
```

```
String::String(const String& rhs)
{
  if (rhs.value->shareable) {
    value = rhs.value;
    ++value->refCount;
  }

  else {
    value = new StringValue(rhs.value->data);
  }
}
```

```
char& String::operator[](int index)
{
  if (value->refCount > 1) {
    --value->refCount;
    value = new StringValue(value->data);
  }

  value->shareable = false;     // add this

  return value->data[index];
}
```

```
class RCObject {
public:
  RCObject();
  RCObject(const RCObject& rhs);
  RCObject& operator=(const RCObject& rhs);
  virtual ~RCObject() = 0;

  void addReference();
  void removeReference();

  void markUnshareable();
  bool isShareable() const;

  bool isShared() const;

private:
  size_t refCount;
  bool shareable;
};
```

```
RCObject::RCObject()
: refCount(0), shareable(true) {}

RCObject::RCObject(const RCObject&)
: refCount(0), shareable(true) {}

RCObject& RCObject::operator=(const RCObject&)
{ return *this; }

RCObject::~RCObject() {}          // virtual dtors must always
                                  // be implemented, even if
                                  // they are pure virtual
                                  // and do nothing (see also
                                  // Item 33)

void RCObject::addReference() { ++refCount; }

void RCObject::removeReference()
{ if (--refCount == 0) delete this; }

void RCObject::markUnshareable()
{ shareable = false; }

bool RCObject::isShareable() const
{ return shareable; }

bool RCObject::isShared() const
{ return refCount > 1; }
```

```
sv1 = sv2;                       // how are sv1's and sv2's reference
                                 // counts affected?
```

```
class String {
private:
  struct StringValue: public RCObject {
    char *data;

    StringValue(const char *initValue);
    ~StringValue();

  };

...

};

String::StringValue::StringValue(const char *initValue)
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}

String::StringValue::~StringValue()
{
  delete [] data;
}
```

```
class String {
private:
  struct StringValue: public RCObject { ... };

  StringValue *value;             // value of this String

  ...

};
```

```
// template class for smart pointers-to-T objects. T must
// support the RCObject interface, typically by inheriting
// from RCObject
template<class T>
class RCPtr {
public:
  RCPtr(T* realPtr = 0);
  RCPtr(const RCPtr& rhs);
  ~RCPtr();

  RCPtr& operator=(const RCPtr& rhs);

  T* operator->() const;        // see Item 28
  T& operator*() const;         // see Item 28

private:
  T *pointee;                   // dumb pointer this
                                // object is emulating

  void init();                  // common initialization
};                              // code
```

```
template<class T>
RCPtr<T>::RCPtr(T* realPtr): pointee(realPtr)
{
  init();
}

template<class T>
RCPtr<T>::RCPtr(const RCPtr& rhs): pointee(rhs.pointee)
{
  init();
}
template<class T>
void RCPtr<T>::init()
{
  if (pointee == 0) {              // if the dumb pointer is
    return;                        // null, so is the smart one
  }

  if (pointee->isShareable() == false) {  // if the value
    pointee = new T(*pointee);            // isn't shareable,
  }                                       // copy it

  pointee->addReference();         // note that there is now a
}                                  // new reference to the value
```

```
class String {
private:

  struct StringValue: public RCObject {
    StringValue(const StringValue& rhs);

    ...

  };

  ...

};

String::StringValue::StringValue(const StringValue& rhs)
{
  data = new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);
}
```

```
class String {
private:
  struct StringValue: public RCObject { ... };

  struct SpecialStringValue: public StringValue { ... };

  ...

};
```

```
pointee = new T(*pointee);      // T is StringValue, but
                                // pointee really points to
                                // a SpecialStringValue
```

```
template<class T>
RCPtr<T>& RCPtr<T>::operator=(const RCPtr& rhs)
{
  if (pointee != rhs.pointee) {    // skip assignments
                                   // where the value
                                   // doesn't change

    T *oldPointee = pointee;       // save old pointee value

    pointee = rhs.pointee;         // point to new value
    init();                        // if possible, share it
                                   // else make own copy

    if (oldPointee) {
      oldPointee->removeReference();// remove reference to
    }                              // current value

  return *this;
}
```

```
template<class T>
RCPtr<T>::~RCPtr()
{
  if (pointee) pointee->removeReference();
}
```

```
template<class T>
T* RCPtr<T>::operator->() const { return pointee; }

template<class T>
T& RCPtr<T>::operator*() const { return *pointee; }
```

```
template<class T>                   // template class for smart
class RCPtr {                       // pointers-to-T objects; T
public:                             // must inherit from RCObject
  RCPtr(T* realPtr = 0);
  RCPtr(const RCPtr& rhs);
  ~RCPtr();

  RCPtr& operator=(const RCPtr& rhs);

  T* operator->() const;
  T& operator*() const;

private:
  T *pointee;

  void init();
};
class RCObject {                     // base class for reference-
public:                              // counted objects
  RCObject();
  RCObject(const RCObject& rhs);
  RCObject& operator=(const RCObject& rhs);
  virtual ~RCObject() = 0;

  void addReference();
  void removeReference();

  void markUnshareable();
  bool isShareable() const;

  bool isShared() const;
private:
  size_t refCount;
  bool shareable;
};

class String {                        // class to be used by
public:                               // application developers

  String(const char *value = "");

  const char& operator[](int index) const;
  char& operator[](int index);

private:
  // class representing string values
  struct StringValue: public RCObject {
    char *data;

    StringValue(const char *initValue);
    StringValue(const StringValue& rhs);
    void init(const char *initValue);
    ~StringValue();
  };

  RCPtr<StringValue> value;
};
```

```
RCObject::RCObject()
: refCount(0), shareable(true) {}

RCObject::RCObject(const RCObject&)
: refCount(0), shareable(true) {}

RCObject& RCObject::operator=(const RCObject&)
{ return *this; }

RCObject::~RCObject() {}

void RCObject::addReference() { ++refCount; }

void RCObject::removeReference()
{ if (--refCount == 0) delete this; }

void RCObject::markUnshareable()
{ shareable = false; }

bool RCObject::isShareable() const
{ return shareable; }

bool RCObject::isShared() const
{ return refCount > 1; }
```

```
template<class T>
void RCPtr<T>::init()
{
  if (pointee == 0) return;

  if (pointee->isShareable() == false) {
    pointee = new T(*pointee);
  }

  pointee->addReference();
}

template<class T>
RCPtr<T>::RCPtr(T* realPtr)
: pointee(realPtr)
{ init(); }

template<class T>
RCPtr<T>::RCPtr(const RCPtr& rhs)
: pointee(rhs.pointee)
{ init(); }

template<class T>
RCPtr<T>::~RCPtr()
{ if (pointee) pointee->removeReference(); }

template<class T>
RCPtr<T>& RCPtr<T>::operator=(const RCPtr& rhs)
{
  if (pointee != rhs.pointee) {
    T *oldPointee = pointee;
    pointee = rhs.pointee;
    init();
    if (oldPointee) oldPointee->removeReference();
  }

  return *this;
}

template<class T>
T* RCPtr<T>::operator->() const { return pointee; }

template<class T>
T& RCPtr<T>::operator*() const { return *pointee; }
```

```
void String::StringValue::init(const char *initValue)
{
 data = new char[strlen(initValue) + 1];
 strcpy(data, initValue);
}

String::StringValue::StringValue(const char *initValue)
{ init(initValue); }

String::StringValue::StringValue(const StringValue& rhs)
{ init(rhs.data); }

String::StringValue::~StringValue()
{ delete [] data; }
```

```
String::String(const char *initValue)
: value(new StringValue(initValue)) {}

const char& String::operator[](int index) const
{ return value->data[index]; }

char& String::operator[](int index)
{
  if (value->isShared()) {
    value = new StringValue(value->data);
  }

  value->markUnshareable();

  return value->data[index];
}
```

```
template<class T>
class RCIPtr {
public:
  RCIPtr(T* realPtr = 0);
  RCIPtr(const RCIPtr& rhs);
  ~RCIPtr();

  RCIPtr& operator=(const RCIPtr& rhs);

  T* operator->() const;
  T& operator*() const;

  RCObject& getRCObject()         // give clients access to
  { return *counter; }            // isShared, etc.

private:
  struct CountHolder: public RCObject {
    ~CountHolder() { delete pointee; }
    T *pointee;
  };

  CountHolder *counter;

  void init();
};

template<class T>
void RCIPtr<T>::init()
{
  if (counter->isShareable() == false) {
    T *oldValue = counter->pointee;
    counter = new CountHolder;
    counter->pointee = oldValue ? new T(*oldValue) : 0;
  }

  counter->addReference();
}

template<class T>
RCIPtr<T>::RCIPtr(T* realPtr)
: counter(new CountHolder)
{
  counter->pointee = realPtr;
  init();
}

template<class T>
RCIPtr<T>::RCIPtr(const RCIPtr& rhs)
: counter(rhs.counter)
{ init(); }

template<class T>
RCIPtr<T>::~RCIPtr()
{ counter->removeReference(); }

template<class T>
RCIPtr<T>& RCIPtr<T>::operator=(const RCIPtr& rhs)
{
  if (counter != rhs.counter) {
    counter->removeReference();
    counter = rhs.counter;
    init();
  }

  return *this;
}

template<class T>
T* RCIPtr<T>::operator->() const
{ return counter->pointee; }

template<class T>
T& RCIPtr<T>::operator*() const
{ return *(counter->pointee); }
```

```
class Widget {
public:
  Widget(int size);
  Widget(const Widget& rhs);
  ~Widget();
  Widget& operator=(const Widget& rhs);
  void doThis();
  int showThat() const;
};
```

```
class RCWidget {
public:
  RCWidget(int size): value(new Widget(size)) {}
  void doThis()
  {
    if (value.getRCObject().isShared()) { // do COW if
      value = new Widget(*value);         // Widget is shared
    }
    value->doThis();
  }
  int showThat() const { return value->showThat(); }
private:
  RCIPtr<Widget> value;
};
```

```
void processInput(int dim1, int dim2)
{
  int data[dim1][dim2];          // error! array dimensions
  ...                            // must be known during
}                                // compilation
```

```
int *data =
  new int[dim1][dim2];            // error!
```

```
Array2D<int> data(10, 20);              // fine

Array2D<float> *data =
  new Array2D<float>(10, 20);           // fine

void processInput(int dim1, int dim2)
{
  Array2D<int> data(dim1, dim2);        // fine
  ...
}
```

```
template<class T>
class Array2D {
public:

  // declarations that won’t compile
  T& operator[][](int index1, int index2);
  const T& operator[][](int index1, int index2) const;

  ...

};
```

```
template<class T>
class Array2D {
public:

  // declarations that will compile
  T& operator()(int index1, int index2);
  const T& operator()(int index1, int index2) const;

  ...

};
```

```
template<class T>
class Array2D {
public:
  class Array1D {
  public:
    T& operator[](int index);
    const T& operator[](int index) const;
    ...

  };

  Array1D operator[](int index);
  const Array1D operator[](int index) const;
  ...

};
```

```
Array2D<float> data(10, 20);
...
cout << data[3][6];                // fine
```

```
String s1, s2;                   // a string-like class; the
                                 // use of proxies keeps this
                                 // class from conforming to
                                 // the standard string
...                              // interface

cout << s1[5];                   // read s1

s2[5] = 'x';                     // write s2

s1[3] = s2[8];                   // write s1, read s2
```

```
class String {
public:
  const char& operator[](int index) const;     // for reads
  char& operator[](int index);                 // for writes
  ...

};
```

```
String s1, s2;

...

cout << s1[5];                  // calls non-const operator[],
                                // because s1 isn’t const

s2[5] = ’x’;                    // also calls non-const
                                // operator[]: s2 isn’t const

s1[3] = s2[8];                  // both calls are to non-const
                                // operator[], because both s1
                                // and s2 are non-const objects
```

```
class String {             // reference-counted strings;
public:                    // see Item 29 for details

  class CharProxy {        // proxies for string chars
  public:
    CharProxy(String& str, int index);          // creation

    CharProxy& operator=(const CharProxy& rhs); // lvalue
    CharProxy& operator=(char c);               // uses

    operator char() const;                      // rvalue
                                                // use
  private:
    String& theString;     // string this proxy pertains to

    int charIndex;         // char within that string
                           // this proxy stands for
  };

  // continuation of String class
  const CharProxy
    operator[](int index) const;     // for const Strings

  CharProxy operator[](int index);    // for non-const Strings
  ...

friend class CharProxy;

private:
  RCPtr<StringValue> value;
};
```

```
String s1, s2;                // reference-counted strings
                              // using proxies
...
cout << s1[5];                // still legal, still works
s2[5] = ’x’;                  // also legal, also works
s1[3] = s2[8];                // of course it’s legal,
                              // of course it works
```

```
const String::CharProxy String::operator[](int index) const
{
 return CharProxy(const_cast<String&>(*this), index);
}

String::CharProxy String::operator[](int index)
{
 return CharProxy(*this, index);
}
```

```
String::CharProxy::CharProxy(String& str, int index)
: theString(str), charIndex(index) {}
```

```
String::CharProxy::operator char() const
{
  return theString.value->data[charIndex];
}
```

```
String::CharProxy&
String::CharProxy::operator=(const CharProxy& rhs)
{
  // if the string is sharing a value with other String objects,
  // break off a separate copy of the value for this string only
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
  }

  // now make the assignment: assign the value of the char
  // represented by rhs to the char represented by *this
  theString.value->data[charIndex] =
    rhs.theString.value->data[rhs.charIndex];

  return *this;
}
```

```
String::CharProxy& String::CharProxy::operator=(char c)
{
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
  }

  theString.value->data[charIndex] = c;

  return *this;
}
```

```
class String {
public:

  class CharProxy {
  public:
    ...
    char * operator&();
    const char * operator&() const;
    ...
  };

  ...
};
```

```
const char * String::CharProxy::operator&() const
{
  return &(theString.value->data[charIndex]);
}
```

```
char * String::CharProxy::operator&()
{
  // make sure the character to which this function returns
  // a pointer isn't shared by any other String objects
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
  }

  // we don't know how long the pointer this function
  // returns will be kept by clients, so the StringValue
  // object can never be shared
  theString.value->markUnshareable();

  return &(theString.value->data[charIndex]);
}
```

```
template<class T>              // reference-counted array
class Array {                  // using proxies
public:
  class Proxy {
  public:
    Proxy(Array<T>& array, int index);
    Proxy& operator=(const T& rhs);
    operator T() const;
    ...
  };

  const Proxy operator[](int index) const;
  Proxy operator[](int index);
  ...
};
```

```
Array<int> intArray;
...
intArray[5] = 22;                    // fine
intArray[5] += 5;                    // error!
++intArray[5];                       // error!
```

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  int numerator() const;
  int denominator() const;
  ...
};

Array<Rational> array;
```

```
cout << array[4].numerator();              // error!
int denom = array[22].denominator();       // error!
```

```
void swap(char& a, char& b);    // swaps the value of a and b
String s = "+C+";               // oops, should be "C++"
swap(s[0], s[1]);               // this should fix the
                                // problem, but it won’t
                                // compile
```

```
class TVStation {
public:
  TVStation(int channel);
  ...

};
```

```
watchTV(10, 2.5);                   // watch channel 10 for
                                    // 2.5 hours
```

```
Array<int> intArray;

intArray[4] = 10;

watchTV(intArray[4], 2.5);      // error! no conversion
                                // from Proxy<int> to
                                // TVStation
```

```
class GameObject { ... };
class SpaceShip: public GameObject { ... };
class SpaceStation: public GameObject { ... };
class Asteroid: public GameObject { ... };
```

```
void checkForCollision(GameObject& object1,
                       GameObject& object2)
{
  if (theyJustCollided(object1, object2)) {
    processCollision(object1, object2);
  }
  else {
    ...
  }
}
```

```
class GameObject {
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};

class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);
  ...
};
```

```
// if we collide with an object of unknown type, we
// throw an exception of this type:
class CollisionWithUnknownObject {
public:
  CollisionWithUnknownObject(GameObject& whatWeHit);
  ...
};

void SpaceShip::collide(GameObject& otherObject)
{
  const type_info& objectType = typeid(otherObject);

  if (objectType == typeid(SpaceShip)) {
    SpaceShip& ss = static_cast<SpaceShip&>(otherObject);

    process a SpaceShip-SpaceShip collision;

  }

  else if (objectType == typeid(SpaceStation)) {
    SpaceStation& ss =
      static_cast<SpaceStation&>(otherObject);

    process a SpaceShip-SpaceStation collision;

  }

  else if (objectType == typeid(Asteroid)) {
    Asteroid& a = static_cast<Asteroid&>(otherObject);

    process a SpaceShip-Asteroid collision;

  }

  else {
    throw CollisionWithUnknownObject(otherObject);
  }
}
```

```
class SpaceShip;               // forward declarations
class SpaceStation;
class Asteroid;
class GameObject {
public:
  virtual void collide(GameObject&   otherObject) = 0;
  virtual void collide(SpaceShip&    otherObject) = 0;
  virtual void collide(SpaceStation& otherObject) = 0;
  virtual void collide(Asteroid&     otherobject) = 0;
  ...
};

class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject&   otherObject);
  virtual void collide(SpaceShip&    otherObject);
  virtual void collide(SpaceStation& otherObject);
  virtual void collide(Asteroid&     otherobject);
  ...
};
```

```
void SpaceShip::collide(GameObject& otherObject)
{
  otherObject.collide(*this);
}
```

```
void SpaceShip::collide(SpaceShip& otherObject)
{
 process a SpaceShip-SpaceShip collision;
}

void SpaceShip::collide(SpaceStation& otherObject)
{
 process a SpaceShip-SpaceStation collision;
}

void SpaceShip::collide(Asteroid& otherObject)
{
 process a SpaceShip-Asteroid collision;
}
```

```
class GameObject {
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};

class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);
  virtual void hitSpaceShip(SpaceShip& otherObject);
  virtual void hitSpaceStation(SpaceStation& otherObject);
  virtual void hitAsteroid(Asteroid& otherobject);
  ...
};
void SpaceShip::hitSpaceShip(SpaceShip& otherObject)
{
  process a SpaceShip-SpaceShip collision;
}

void SpaceShip::hitSpaceStation(SpaceStation& otherObject)
{
  process a SpaceShip-SpaceStation collision;
}

void SpaceShip::hitAsteroid(Asteroid& otherObject)
{
  process a SpaceShip-Asteroid collision;
}
```

```
class SpaceShip: public GameObject {
private:
  typedef void (SpaceShip::*HitFunctionPtr)(GameObject&);

  static HitFunctionPtr lookup(const GameObject& whatWeHit);

  ...
};
```

```
void SpaceShip::collide(GameObject& otherObject)
{
  HitFunctionPtr hfp =
    lookup(otherObject);          // find the function to call

  if (hfp) {                      // if a function was found
    (this->*hfp)(otherObject);    // call it
  }
  else {
    throw CollisionWithUnknownObject(otherObject);
  }
}
```

```
class SpaceShip: public GameObject {
private:
  typedef void (SpaceShip::*HitFunctionPtr)(GameObject&);
  typedef map<string, HitFunctionPtr> HitMap;
  ...
};

SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap;
  ...
}
```

```
SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap;    // we’ll see how to
                                 // initialize this below

  // look up the collision-processing function for the type
  // of whatWeHit. The value returned is a pointer-like
  // object called an "iterator" (see Item 35).
  HitMap::iterator mapEntry=
    collisionMap.find(typeid(whatWeHit).name());

  // mapEntry == collisionMap.end() if the lookup failed;
  // this is standard map behavior. Again, see Item 35.
  if (mapEntry == collisionMap.end()) return 0;
  // If we get here, the search succeeded. mapEntry
  // points to a complete map entry, which is a
  // (string, HitFunctionPtr) pair. We want only the
  // second part of the pair, so that’s what we return.
  return (*mapEntry).second;
}
```

```
// An incorrect implementation
SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap;

  collisionMap["SpaceShip"] = &hitSpaceShip;
  collisionMap["SpaceStation"] = &hitSpaceStation;
  collisionMap["Asteroid"] = &hitAsteroid;

  ...

}
```

```
class SpaceShip: public GameObject {
private:
  static HitMap initializeCollisionMap();
  ...

};

SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap = initializeCollisionMap();
  ...
}
```

```
class SpaceShip: public GameObject {
private:
  static HitMap * initializeCollisionMap();

  ...

};

SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static auto_ptr<HitMap>
    collisionMap(initializeCollisionMap());
  ...

}
```

```
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;

  (*phm)["SpaceShip"] = &hitSpaceShip;
  (*phm)["SpaceStation"] = &hitSpaceStation;
  (*phm)["Asteroid"] = &hitAsteroid;

  return phm;
}
```

```
// A bad idea...
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;

  (*phm)["SpaceShip"] =
    reinterpret_cast<HitFunctionPtr>(&hitSpaceShip);

  (*phm)["SpaceStation"] =
    reinterpret_cast<HitFunctionPtr>(&hitSpaceStation);

  (*phm)["Asteroid"] =
    reinterpret_cast<HitFunctionPtr>(&hitAsteroid);

  return phm;
}
```

```
class GameObject {                          // this is unchanged
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};
class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);

  // these functions now all take a GameObject parameter
  virtual void hitSpaceShip(GameObject& spaceShip);
  virtual void hitSpaceStation(GameObject& spaceStation);
  virtual void hitAsteroid(GameObject& asteroid);
  ...
};
```

```
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;

  (*phm)["SpaceShip"] = &hitSpaceShip;
  (*phm)["SpaceStation"] = &hitSpaceStation;
  (*phm)["Asteroid"] = &hitAsteroid;

  return phm;
}
```

```
void SpaceShip::hitSpaceShip(GameObject& spaceShip)
{
  SpaceShip& otherShip=
    dynamic_cast<SpaceShip&>(spaceShip);

  process a SpaceShip-SpaceShip collision;

}

void SpaceShip::hitSpaceStation(GameObject& spaceStation)
{
  SpaceStation& station=
    dynamic_cast<SpaceStation&>(spaceStation);

  process a SpaceShip-SpaceStation collision;

}

void SpaceShip::hitAsteroid(GameObject& asteroid)
{
  Asteroid& theAsteroid =
    dynamic_cast<Asteroid&>(asteroid);

  process a SpaceShip-Asteroid collision;

}
```

```
#include "SpaceShip.h"
#include "SpaceStation.h"
#include "Asteroid.h"

namespace {                // unnamed namespace — see below

  // primary collision-processing functions
  void shipAsteroid(GameObject& spaceShip,
                    GameObject& asteroid);

  void shipStation(GameObject& spaceShip,
                   GameObject& spaceStation);

  void asteroidStation(GameObject& asteroid,
                       GameObject& spaceStation);
  ...

  // secondary collision-processing functions that just
  // implement symmetry: swap the parameters and call a
  // primary function
  void asteroidShip(GameObject& asteroid,
                    GameObject& spaceShip)
  { shipAsteroid(spaceShip, asteroid); }

  void stationShip(GameObject& spaceStation,
                   GameObject& spaceShip)
  { shipStation(spaceShip, spaceStation); }

  void stationAsteroid(GameObject& spaceStation,
                       GameObject& asteroid)
  { asteroidStation(asteroid, spaceStation); }

  ...
  // see below for a description of these types/functions
  typedef void (*HitFunctionPtr)(GameObject&, GameObject&);
  typedef map< pair<string,string>, HitFunctionPtr > HitMap;

  pair<string,string> makeStringPair(const char *s1,
                                     const char *s2);

  HitMap * initializeCollisionMap();

  HitFunctionPtr lookup(const string& class1,
                        const string& class2);

} // end namespace

void processCollision(GameObject& object1,
                      GameObject& object2)
{
  HitFunctionPtr phf = lookup( typeid(object1).name(),
                               typeid(object2).name());

  if (phf) phf(object1, object2);
  else throw UnknownCollision(object1, object2);
}
```

```
// we use this function to create pair<string,string>
// objects from two char* literals. It's used in
// initializeCollisionMap below. Note how this function
// enables the return value optimization (see Item 20).

namespace {      // unnamed namespace again — see below

  pair<string,string> makeStringPair(const char *s1,
                                     const char *s2)
  { return pair<string,string>(s1, s2); }

} // end namespace

namespace {      // still the unnamed namespace — see below

  HitMap * initializeCollisionMap()
  {
    HitMap *phm = new HitMap;

    (*phm)[makeStringPair("SpaceShip","Asteroid")] =
      &shipAsteroid;

    (*phm)[makeStringPair("SpaceShip", "SpaceStation")] =
      &shipStation;

    ...

    return phm;
  }

} // end namespace
```

```
namespace {      // I explain this below — trust me

  HitFunctionPtr lookup(const string& class1,
                        const string& class2)
  {
    static auto_ptr<HitMap>
      collisionMap(initializeCollisionMap());

    // see below for a description of make_pair
    HitMap::iterator mapEntry=
      collisionMap->find(make_pair(class1, class2));

    if (mapEntry == collisionMap->end()) return 0;

    return (*mapEntry).second;
  }

} // end namespace
```

```
HitMap::iterator mapEntry=
  collisionMap->find(make_pair(class1, class2));
```

```
HitMap::iterator mapEntry=
  collisionMap->find(pair<string,string>(class1, class2));
```

```
void shipAsteroid(GameObject& spaceShip,
                  GameObject& asteroid);
```

```
class CollisionMap {
public:
  typedef void (*HitFunctionPtr)(GameObject&, GameObject&);

  void addEntry(const string& type1,
                const string& type2,
                HitFunctionPtr collisionFunction,
                bool symmetric = true);         // see below

  void removeEntry(const string& type1,
                   const string& type2);

  HitFunctionPtr lookup(const string& type1,
                        const string& type2);

  // this function returns a reference to the one and only
  // map — see Item 26
  static CollisionMap& theCollisionMap();

private:
  // these functions are private to prevent the creation
  // of multiple maps — see Item 26
  CollisionMap();
  CollisionMap(const CollisionMap&);
};
```

```
void shipAsteroid(GameObject& spaceShip,
                  GameObject& asteroid);
CollisionMap::theCollisionMap().addEntry("SpaceShip",
                                         "Asteroid",
                                         &shipAsteroid);

void shipStation(GameObject& spaceShip,
                 GameObject& spaceStation);
CollisionMap::theCollisionMap().addEntry("SpaceShip",
                                         "SpaceStation",
                                         &shipStation);

void asteroidStation(GameObject& asteroid,
                     GameObject& spaceStation);
CollisionMap::theCollisionMap().addEntry("Asteroid",
                                         "SpaceStation",
                                         &asteroidStation);
...
```

```
class RegisterCollisionFunction {
public:
  RegisterCollisionFunction(
            const string& type1,
            const string& type2,
            CollisionMap::HitFunctionPtr collisionFunction,
            bool symmetric = true)
  {
    CollisionMap::theCollisionMap().addEntry(type1, type2,
                                             collisionFunction,
                                             symmetric);
  }
};
```

```
RegisterCollisionFunction cf1("SpaceShip", "Asteroid",
                              &shipAsteroid);

RegisterCollisionFunction cf2("SpaceShip", "SpaceStation",
                              &shipStation);

RegisterCollisionFunction cf3("Asteroid", "SpaceStation",
                              &asteroidStation);

...

int main(int argc, char * argv[])
{
  ...
}
```

```
void satelliteShip(GameObject& satellite,
                   GameObject& spaceShip);
void satelliteAsteroid(GameObject& satellite,
                       GameObject& asteroid);
```

```
RegisterCollisionFunction cf4("Satellite", "SpaceShip",
                              &satelliteShip);
RegisterCollisionFunction cf5("Satellite", "Asteroid",
                              &satelliteAsteroid);
```

```
class B { ... };                // no virtual dtor needed
class D: public B { ... };

B *pb = new D;
```

```
delete pb;                          // NOW you need the virtual
                                    // destructor in B
```

```
class string {                      // from the standard C++ library
public:
  ~string();
};

class B { ... };                    // no data members with dtors,
                                    // no virtual dtor needed
```

```
class D: public B {
  string name;                   // NOW ~B needs to be virtual
};
```

```
auto_ptr<String> ps(String::makeString("Future tense C++"));
...                               // treat ps as a pointer to
                                  // a String object, but don't
                                  // worry about deleting it
```

```
class Animal {
public:
  Animal& operator=(const Animal& rhs);
  ...
};
class Lizard: public Animal {
public:
  Lizard& operator=(const Lizard& rhs);
  ...
};

class Chicken: public Animal {
public:
  Chicken& operator=(const Chicken& rhs);
  ...
};
```

```
Lizard liz1;
Lizard liz2;

Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;

...

*pAnimal1 = *pAnimal2;
```

```
class Animal {
public:
  virtual Animal& operator=(const Animal& rhs);
  ...
};
class Lizard: public Animal {
public:
  virtual Lizard& operator=(const Animal& rhs);
  ...
};
class Chicken: public Animal {
public:
  virtual Chicken& operator=(const Animal& rhs);
  ...
};
```

```
Lizard liz;
Chicken chick;

Animal *pAnimal1 = &liz;
Animal *pAnimal2 = &chick;

...

*pAnimal1 = *pAnimal2;          // assign a chicken to
                                // a lizard!
```

```
Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;

...

*pAnimal1 = *pAnimal2;          // assign a lizard to a lizard
```

```
Animal *pAnimal1 = &liz;
Animal *pAnimal2 = &chick;

...

*pAnimal1 = *pAnimal2;        // assign a chicken to a lizard
```

```
Lizard& Lizard::operator=(const Animal& rhs)
{
  // make sure rhs is really a lizard
  const Lizard& rhs_liz = dynamic_cast<const Lizard&>(rhs);

  proceed with a normal assignment of rhs_liz to *this;

}
```

```
Lizard liz1, liz2;
...
liz1 = liz2;                          // no need to perform a
                                      // dynamic_cast: this
                                      // assignment must be valid
```

```
class Lizard: public Animal {
public:
  virtual Lizard& operator=(const Animal& rhs);
  Lizard& operator=(const Lizard& rhs);         // add this
  ...
};
Lizard liz1, liz2;
...
liz1 = liz2;                    // calls operator= taking
                                // a const Lizard&
Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;
...
*pAnimal1 = *pAnimal2;          // calls operator= taking
                                // a const Animal&
```

```
Lizard& Lizard::operator=(const Animal& rhs)
{
  return operator=(dynamic_cast<const Lizard&>(rhs));
}
```

```
class Animal {
private:
  Animal& operator=(const Animal& rhs);     // this is now
  ...                                       // private

};

class Lizard: public Animal {
public:
  Lizard& operator=(const Lizard& rhs);
  ...

};

class Chicken: public Animal {
public:
  Chicken& operator=(const Chicken& rhs);
  ...

};

Lizard liz1, liz2;
...
liz1 = liz2;                     // fine

Chicken chick1, chick2;
...
chick1 = chick2;                 // also fine

Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &chick1;
...
*pAnimal1 = *pAnimal2;           // error! attempt to call
                                 // private Animal::operator=
Animal animal1, animal2;
...
animal1 = animal2;                  // error! attempt to call
                                    // private Animal::operator=
Lizard& Lizard::operator=(const Lizard& rhs)
{
  if (this == &rhs) return *this;

  Animal::operator=(rhs);       // error! attempt to call
                                // private function. But
                                // Lizard::operator= must
                                // call this function to
  ...                           // assign the Animal parts
}                               // of *this!
```

```
class AbstractAnimal {
protected:
  AbstractAnimal& operator=(const AbstractAnimal& rhs);

public:
  virtual ~AbstractAnimal() = 0;                 // see below
  ...

};

class Animal: public AbstractAnimal {
public:
  Animal& operator=(const Animal& rhs);
  ...
};

class Lizard: public AbstractAnimal {
public:
  Lizard& operator=(const Lizard& rhs);
  ...
};

class Chicken: public AbstractAnimal {
public:
  Chicken& operator=(const Chicken& rhs);
  ...
};
```

```
class Window {                  // this is the library class
public:
  virtual void resize(int newWidth, int newHeight);
  virtual void repaint() const;

  int width() const;
  int height() const;
};

class SpecialWindow {           // this is the class you
public:                         // wanted to have inherit
  ...                           // from Window

  // pass-through implementations of nonvirtual functions
  int width() const { return w.width(); }
  int height() const { return w.height(); }

  // new implementations of "inherited" virtual functions
  virtual void resize(int newWidth, int newHeight);
  virtual void repaint() const;

private:
  Window w;
};
```

```
// declare a function called drawLine; don’t mangle
// its name
extern "C"
void drawLine(int x1, int y1, int x2, int y2);
```

```
// this function is in assembler — don’t mangle its name
extern "C" void twiddleBits(unsigned char bits);
```

```
// the following C++ function is designed for use outside
// C++ and should not have its name mangled
extern "C" void simulate(int iterations);
extern "C" {                // disable name mangling for
                            // all the following functions

  void drawLine(int x1, int y1, int x2, int y2);
  void twiddleBits(unsigned char bits);
  void simulate(int iterations);
  ...

}
```

```
#ifdef __cplusplus

extern "C" {

#endif

  void drawLine(int x1, int y1, int x2, int y2);
  void twiddleBits(unsigned char bits);
  void simulate(int iterations);
  ...

#ifdef __cplusplus

}

#endif
```

```
int main(int argc, char *argv[])
{
  performStaticInitialization();      // generated by the
                                      // implementation

  the statements you put in main go here;

  performStaticDestruction();         // generated by the
                                      // implementation
}
```

```
extern "C"                                 // implement this
int realMain(int argc, char *argv[]);   // function in C

int main(int argc, char *argv[])         // write this in C++
{
  return realMain(argc, argv);
}
```

```
char * strdup(const char *ps);    // return a copy of the
                                  // string pointed to by ps
```

```
template<class charT,
           class traits = char_traits<charT>,
           class Allocator = allocator<charT> >
  class basic_string;
```

```
int * find(int *begin, int *end, int value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

```
int values[50];

...

int *firstFive = find(values,     // search the range
                      values+50,  // values[0] - values[49]
                      5);         // for the value 5

if (firstFive != values+50) {     // did the search succeed?

  ...                             // yes

}
else {
  ...                             // no, the search failed
}
```

```
int *firstFive = find(values,      // search the range
                      values+10,   // values[0] - values[9]
                      5);          // for the value 5

int age = 36;

...

int *firstValue = find(values+10,  // search the range
                       values+20,  // values[10] - values[19]
                       age);       // for the value in age
```

```
template<class T>
T * find(T *begin, T *end, const T& value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

```
template<class Iterator, class T>
Iterator find(Iterator begin, Iterator end, const T& value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

```
list<char> charList;             // create STL list object
                                 // for holding chars
...
// find the first occurrence of 'x' in charList
list<char>::iterator it = find(charList.begin(),
                               charList.end(),
                               'x');
```

```
int values[50];
...
int *firstFive = find(values, values+50, 5);  // fine, calls
                                              // STL find
```

```
template<class T>
class auto_ptr {
public:
  explicit auto_ptr(T *p = 0);  // see Item 5 for a
                                // description of "explicit"

  template<class U>             // copy constructor member
  auto_ptr(auto_ptr<U>& rhs);   // template (see Item 28):
                                // initialize a new auto_ptr
                                // with any compatible
                                // auto_ptr

  ~auto_ptr();

  template<class U>             // assignment operator
  auto_ptr<T>&                  // member template (see
  operator=(auto_ptr<U>& rhs);  // Item 28): assign from any
                                // compatible auto_ptr

  T& operator*() const;         // see Item 28
  T* operator->() const;        // see Item 28

  T* get() const;               // return value of current
                                // dumb pointer

  T* release();                 // relinquish ownership of
                                // current dumb pointer and
                                // return its value

  void reset(T *p = 0);         // delete owned pointer;
                                // assume ownership of p

private:
  T *pointee;
template<class U>                // make all auto_ptr classes
friend class auto_ptr<U>;        // friends of one another
};

template<class T>
inline auto_ptr<T>::auto_ptr(T *p)
: pointee(p)
{}

template<class T>
  template<class U>
  inline auto_ptr<T>::auto_ptr(auto_ptr<U>& rhs)
  : pointee(rhs.release())
  {}

template<class T>
inline auto_ptr<T>::~auto_ptr()
{ delete pointee; }

template<class T>
  template<class U>
  inline auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<U>& rhs)
  {
    if (this != &rhs) reset(rhs.release());
    return *this;
  }

template<class T>
inline T& auto_ptr<T>::operator*() const
{ return *pointee; }
```

```
template<class T>
inline T* auto_ptr<T>::operator->() const
{ return pointee; }

template<class T>
inline T* auto_ptr<T>::get() const
{ return pointee; }
template<class T>
inline T* auto_ptr<T>::release()
{
  T *oldPointee = pointee;
  pointee = 0;
  return oldPointee;
}

template<class T>
inline void auto_ptr<T>::reset(T *p)
{
  if (pointee != p) {
    delete pointee;
    pointee = p;
  }
}
```

```
template<class T>
class auto_ptr {
public:
  explicit auto_ptr(T *p = 0): pointee(p) {}

  template<class U>
  auto_ptr(auto_ptr<U>& rhs): pointee(rhs.release()) {}

  ~auto_ptr() { delete pointee; }

  template<class U>
  auto_ptr<T>& operator=(auto_ptr<U>& rhs)
  {
    if (this != &rhs) reset(rhs.release());
    return *this;
  }

  T& operator*() const { return *pointee; }

  T* operator->() const { return pointee; }

  T* get() const { return pointee; }

  T* release()
  {
    T *oldPointee = pointee;
    pointee = 0;
    return oldPointee;
  }

  void reset(T *p = 0)
  {
    if (pointee != p) {
      delete pointee;
      pointee = p;
    }
  }
 private:
   T *pointee;

 template<class U> friend class auto_ptr<U>;
  };
```

```
template<typename C>
bool lastGreaterThanFirst(const C& container)
{
   if (container.empty()) return false;

   typename C::const_iterator begin(container.begin());
   typename C::const_iterator end(container.end());

   return *--end > *begin;
}
```

```
class Widget { ... };
bool operator==(const Widget& lhs, const Widget& rhs);
```

```
class Widget { ... };
vector<Widget> vw;
Widget bestWidget;
...                                         // give bestWidget a value
vector<Widget>::iterator i =                // find a Widget with the
   find(vw.begin(), vw.end(), bestWidget);  // same value as bestWidget
```

```
class Widget { ... };
typedef vector<Widget> WidgetContainer;
WidgetContainer cw;
Widget bestWidget;
...
WidgetContainer::iterator i = find(cw.begin(), cw.end(), bestWidget);
```

```
class Widget { ... };

template<typename T>                        // see Item 10 for why this
SpecialAllocator { ... };                   // needs to be a template

typedef vector<Widget, SpecialAllocator<Widget> > WidgetContainer;

WidgetContainer cw;                         // still works

Widget bestWidget;

...

WidgetContainer::iterator i =
  find(cw.begin(), cw.end(), bestWidget);   // still works
```

```
map< string,
        vector<Widget>::iterator,
        CIStringCompare>                        // CIStringCompare is "case-
                                                // insensitive string compare;"
                                                // Item 19 describes it
```

```
class CustomerList {
private:
   typedef list<Customer> CustomerContainer;
   typedef CustomerContainer::iterator CCIterator;

   CustomerContainer customers;

public:                                        // limit the amount of list-specific
   ...                                         // information visible through
};                                             // this interface
```

```
class Widget {
public:
   ...
   Widget(const Widget&);                        // copy constructor
   Widget& operator=(const Widget&);             // copy assignment operator
   ...
};
```

```
vector<Widget> vw;

class SpecialWidget:                 // SpecialWidget inherits from
   public Widget { ... };            // Widget above

SpecialWidget sw;

vw.push_back(sw);                    // sw is copied as a base class
                                     // object into vw. Its specialness
                                     // is lost during the copying
```

```
Widget w[maxNumWidgets];    // create an array of maxNumWidgets
                            // Widgets, default-constructing each one
```

```
vector<Widget> vw;                    // create a vector with zero Widget
                                      // objects that will expand as needed
```

```
vector<Widget> vw;
vw.reserve(maxNumWidgets);        // see Item 14 for details on reserve
```

```
list<int> list1;
list<int> list2;

...

list1.splice(                                       // move all nodes in list2
   list1.end(), list2,                              // from the first occurrence
   find(list2.begin(), list2.end(), 5),             // of 5 through the last
   find(list2.rbegin(), list2.rend(), 10).base()    // occurrence of 10 to the
);                                                  // end of list1. See Item 28
                                                    // for info on the "base()" call
```

```
vector<Widget> v1, v2;                    // assume v1 and v2 are vectors
                                          // of Widgets
...
v1.clear();
for (vector<Widget>::const_iterator ci = v2.begin() + v2.size() / 2;
      ci != v2.end();
      ++ci)
   v1.push_back(*ci);
```

```
int data[numValues];                                // assume numValues is
                                                    // defined elsewhere
vector<int> v;
...
v.insert(v.begin(), data, data + numValues);        // insert the ints in data
                                                    // into v at the front
```

```
vector<int>::iterator insertLoc(v.begin());
for (int i = 0; i < numValues; ++i) {
   insertLoc = v.insert(insertLoc, data[i]);
   ++insertLoc;
}
```

```
void container::insert(iterator position,          // where to insert the range
                       InputIterator begin,        // start of range to insert
                       InputIterator end);         // end of range to insert
```

```
ifstream dataFile("ints.dat");
list<int> data(istream_iterator<int>(dataFile),     // warning! this doesn't do
               istream_iterator<int>());            // what you think it does
```

```
class Widget { ... };            // assume Widget has a default constructor
Widget w();                      // uh oh...
```

```
list<int> data((istream_iterator<int>(dataFile)),    // note new parens
                istream_iterator<int>());              // around first argument
                                                       // to list's constructor
```

```
ifstream dataFile("ints.dat");

istream_iterator<int> dataBegin(dataFile);
istream_iterator<int> dataEnd;

list<int> data(dataBegin, dataEnd);
```

```
void doSomething()
{
   vector<Widget*> vwp;
   for (int i = 0; i < SOME_MAGIC_NUMBER; ++i)
      vwp.push_back(new Widget);
   ...                                             // use vwp
}                                                  // Widgets are leaked here!
```

```
void doSomething()
{
   vector<Widget*> vwp;
   ...                                                    // as before
   for( vector<Widget*>::iterator i = vwp.begin();
         i != vwp.end();
         ++i)
      delete *i;
}
```

```
template<typename T>
struct DeleteObject:                              // Item 40 describes why
   public unary_function<const T*, void> {        // this inheritance is here

   void operator()(const T* ptr) const
   {
      delete ptr;
   }

};
```

```
void doSomething()
{
   ...                                              // as before
   for_each(vwp.begin(), vwp.end(), DeleteObject<Widget>());
}
```

```
void doSomething()
{
   deque<SpecialString*> dssp;

   ...
   for_each( dssp.begin(), dssp.end(),          // undefined behavior! Deletion
               DeleteObject<string>());         // of a derived object via a base
}                                               // class pointer where there is
                                                // no virtual destructor
```

```
struct DeleteObject {                          // templatization and base
                                               // class removed here

   template<typename T>                        // templatization added here
   void operator()(const T* ptr) const
   {
      delete ptr;
   }

};
```

```
void doSomething()
{
   deque<SpecialString*> dssp;
   ...
   for_each( dssp.begin(), dssp.end(),
             DeleteObject());                  // ah! well-defined behavior!
}
```

```
void doSomething()
{
  typedef boost::shared_ptr<Widget> SPW;        // SPW = "shared_ptr
                                                //          to Widget"
  vector<SPW> vwp;

  for (int i = 0; i < SOME_MAGIC_NUMBER; ++i)
    vwp.push_back(SPW(new Widget));             // create an SPW from a
                                                // Widget*, then do a
                                                // push_back on it

  ...                                           // use vwp

}                                         // no Widgets are leaked here, not
                                          // even if an exception is thrown
                                          // in the code above
```

```
auto_ptr<Widget> pw1(new Widget);    // pw1 points to a Widget
auto_ptr<Widget> pw2(pw1);           // pw2 points to pw1's Widget;
                                     // pw1 is set to NULL. (Ownership
                                     // of the Widget is transferred
                                     // from pw1 to pw2.)
pw1 = pw2;                           // pw1 now points to the Widget
                                     // again; pw2 is set to NULL
```

```
bool widgetAPCompare(const auto_ptr<Widget>& lhs,
                     const auto_ptr<Widget>& rhs)
{
  return *lhs < *rhs;                 // for this example, assume that
}                                     // operator< exists for Widgets

vector<auto_ptr<Widget> > widgets;    // create a vector and then fill it
...                                   // with auto_ptrs to Widgets;
                                      // remember that this should
                                      // not compile!

sort(widgets.begin(), widgets.end(),  // sort the vector
     widgetAPCompare);
```

```
template<class RandomAccessIterator,          // this declaration for
            class Compare>                    // sort is copied straight
void sort(RandomAccessIterator first,         // out of the Standard
          RandomAccessIterator last,
          Compare comp)
{
  // this typedef is described below
  typedef typename iterator_traits<RandomAccessIterator>::value_type
    ElementType;

  RandomAccessIterator i;

  ...                                 // make i point to the pivot element

  ElementType pivotValue(*i);         // copy the pivot element into a
                                      // local temporary variable; see
                                      // discussion below

  ...                                 // do the rest of the sorting work
}
```

```
c.erase( remove(c.begin(), c.end(), 1963),      // the erase-remove idiom is
         c.end());                               // the best way to get rid of
                                                 // elements with a specific
                                                 // value when c is a vector,
                                                 // string, or deque
```

```
c.remove(1963);                         // the remove member function is the
                                        // best way to get rid of elements with
                                        // a specific value when c is a list
```

```
c.erase(1963);                    // the erase member function is the
                                  // best way to get rid of elements with
                                  // a specific value when c is a
                                  // standard associative container
```

```
c.erase( remove_if(c.begin(), c.end(), badValue),     // this is the best way to
         c.end());                                    // get rid of objects
                                                      // where badValue
                                                      // returns true when c is
                                                      // a vector, string, or
                                                      // deque

c.remove_if(badValue);                       // this is the best way to get rid of
                                             // objects where badValue returns
                                             // true when c is a list
```

```
AssocContainer<int> c;                        // c is now one of the
...                                           // standard associative
                                              // containers

AssocContainer<int> goodValues;               // temporary container
                                              // to hold unremoved
                                              // values

remove_copy_if(c.begin(), c.end(),            // copy desired
               inserter( goodValues,          // values from c to
                         goodValues.end()),   // goodValues
               badValue);

c.swap(goodValues);                           // swap the contents of
                                              // c and goodValues
```

```
AssocContainer<int> c;
...
for (AssocContainer<int>::iterator i = c.begin();    // clear, straightforward,
      i != c.end();                                  // and buggy code to
      ++i) {                                         // erase every element
   if (badValue(*i)) c.erase(i);                     // in c where badValue
}                                                    // returns true; don't
                                                     // do this!
```

```
AssocContainer<int> c;
...
for (AssocContainer<int>::iterator i = c.begin();   // the 3rd part of the for
      i != c.end();                                 // loop is empty; i is now
      /* nothing */) {                              // incremented below
   if (badValue(*i)) c.erase(i++);                  // for bad values, pass the
   else ++i;                                        // current i to erase and
}                                                   // increment i as a side
                                                    // effect; for good values,
                                                    // just increment i
```

```
ofstream logFile;                                   // log file to write to
AssocContainer<int> c;
...
for (AssocContainer<int>::iterator i = c.begin();   // loop conditions are the
      i != c.end(); ) {                             // same as before
   if (badValue(*i)) {
      logFile << "Erasing " << *i << '\n';          // write log file
      c.erase(i++);                                 // erase element
   }
   else ++i;
}
```

```
for (SeqContainer<int>::iterator i = c.begin();
      i != c.end(); ) {
   if (badValue(*i)) {
      logFile << "Erasing " << *i << '\n';
      i = c.erase(i);                                // keep i valid by assigning
   }                                                 // erase's return value to it
   else ++i;
}
```

```
template<typename T>                        // a user-defined allocator
class SpecialAllocator { ... };             // template

typedef SpecialAllocator<Widget> SAW;       // SAW = "SpecialAllocator
                                            //           for Widgets"

list<Widget, SAW> L1;
list<Widget, SAW> L2;

...

L1.splice(L1.begin(), L2);                  // move L2's nodes to the
                                            // front of L1
```

```
void* operator new(size_t bytes);
pointer allocator<T>::allocate(size_type numObjects);
                                          // recall that "pointer" is a typedef
                                          // that's virtually always T*
```

```
list<int> L;                     // same as list<int, allocator<int> >;
                                 // allocator<int> is never asked to
                                 // allocate memory!
set<Widget, SAW> s;              // recall that SAW is a typedef for
                                 // SpecialAllocator<Widget>; no
                                 // SAW will ever allocate memory!
```

```
template<typename T,                                      // possible list
            typename Allocator = allocator<T> >           // implementation
class list {
private:
  Allocator alloc;                        // allocator for objects of type T
  struct ListNode {                       // nodes in the linked list
    T data;
    ListNode *prev;
    ListNode *next;
  };
  ...
};
```

```
template<typename T>                   // the standard allocator is declared
class allocator {                      // like this, but this could be a user-
public:                                // written allocator template, too

  template<typename U>
  struct rebind {
    typedef allocator<U> other;
  };

  ...
};
```

```
template<typename T>
class SharedMemoryAllocator {
public:
   ...
   pointer allocate(size_type numObjects, const void *localityHint = 0)
   {
      return static_cast<pointer>(mallocShared(numObjects * sizeof(T)));
   }

   void deallocate(pointer ptrToMemory, size_type numObjects)
   {
      freeShared(ptrToMemory);
   }

   ...
};
```

```
// convenience typedef
typedef
  vector<double, SharedMemoryAllocator<double> > SharedDoubleVec;
...
{                                       // begin some block
  ...
  SharedDoubleVec v;                    // create a vector whose elements
                                        // are in shared memory

  ...
}                                       // end the block
```

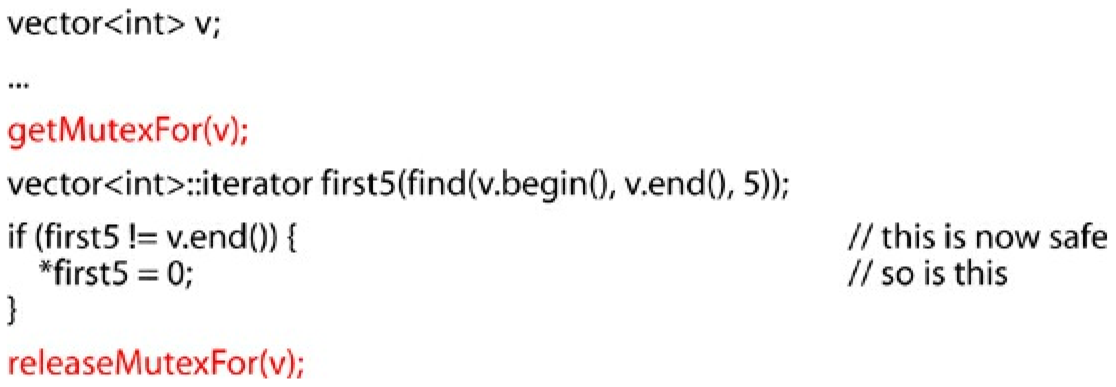
```
void *pVectorMemory =                          // allocate enough shared
  mallocShared(sizeof(SharedDoubleVec));       // memory to hold a
                                               // SharedDoubleVec object

SharedDoubleVec *pv =                          // use “placement new” to
  new (pVectorMemory) SharedDoubleVec;         // create a SharedDoubleVec
                                               // object in the memory;
                                               // see below

...                                            // use the object (via pv)

pv->~SharedDoubleVec();                        // destroy the object in the
                                               // shared memory

freeShared(pVectorMemory);                     // deallocate the initial
                                               // chunk of shared memory
```

```
class Heap1 {
public:
  ...
  static void* alloc(size_t numBytes, const void *memoryBlockToBeNear);
  static void dealloc(void *ptr);
  ...
};

class Heap2 { ... };                    // has the same alloc/dealloc interface
```

```
template<typename T, typename Heap>
class SpecificHeapAllocator {
public:
   ...
   pointer allocate(size_type numObjects, const void *localityHint = 0)
   {
      return static_cast<pointer> (Heap::alloc(numObjects * sizeof(T),
                                   localityHint));
   }

   void deallocate(pointer ptrToMemory, size_type numObjects)
   {
      Heap::dealloc(ptrToMemory);
   }

   ...
};
```

```
vector<int, SpecificHeapAllocator<int, Heap1> > v;       // put both v's and
set<int, SpecificHeapAllocator<int, Heap1> > s;          // s's elements in
                                                         // Heap1

list< Widget,
     SpecificHeapAllocator<Widget, Heap2> > L;           // put both L's and
map< int, string, less<int>,                             // m's elements in
       SpecificHeapAllocator<pair<const int, string>,    // Heap2
                                 Heap2> > m;
```

```
vector<int> v;
...
vector<int>::iterator first5(find(v.begin(), v.end(), 5));     // Line 1
if (first5 != v.end()) {                                       // Line 2
   *first5 = 0;                                                // Line 3
}
```



```
vector<int> v;
...
getMutexFor(v);
vector<int>::iterator first5(find(v.begin(), v.end(), 5));
if (first5 != v.end()) {                                    // this is now safe
    *first5 = 0;                                            // so is this
}
releaseMutexFor(v);
```

```
template<typename Container>          // skeletal template for classes
class Lock {                          // that acquire and release mutexes
public:                               // for containers; many details
                                      // have been omitted
  Lock(const Container& container)
  : c(container)
  {
    getMutexFor(c);                   // acquire mutex in the constructor
  }

  ~Lock()
  {
    releaseMutexFor(c);               // release it in the destructor
  }

private:
  const Container& c;
};
```

```
vector<int> v;
...
{                                               // create new block
   Lock<vector<int> > lock(v);                  // acquire mutex
   vector<int>::iterator first5(find(v.begin(), v.end(), 5));
   if (first5 != v.end()) {
      *first5 = 0;
   }
}                                               // close block, automatically
                                                // releasing the mutex
```

```
vector<int> v;
v.reserve(1000);
for (int i = 1; i <= 1000; ++i) v.push_back(i);
```

```
string s("Perse");                // Our dog is named “Persephone”, but we
                                  // usually just call her “Perse”. Visit her web site
                                  // at http://www.aristeia.com/Persephone/
// C API: this function takes a pointer to an array of at most arraySize
// doubles and writes data to it. It returns the number of doubles written,
// which is never more than arraySize.
size_t fillArray(double *pArray, size_t arraySize);

vector<double> vd(maxNumDoubles);            // create a vector whose
                                             // size is maxNumDoubles

vd.resize(fillArray(&vd[0], vd.size()));     // have fillArray write data
                                             // into vd, then resize vd
                                             // to the number of
                                             // elements fillArray wrote
// C API: this function takes a pointer to an array of at most arraySize
// chars and writes data to it. It returns the number of chars written,
// which is never more than arraySize.
size_t fillString(char *pArray, size_t arraySize);
vector<char> vc(maxNumChars);                         // create a vector whose
                                                      // size is maxNumChars

size_t charsWritten = fillString(&vc[0], vc.size());  // have fillString write
                                                      // into vc

string s(vc.begin(), vc.begin()+charsWritten);        // copy data from vc to s
                                                      // via range constructor
                                                      // (see Item 5)
```

```
size_t fillArray(double *pArray, size_t arraySize);       // as above
vector<double> vd(maxNumDoubles);                         // also as above
vd.resize(fillArray(&vd[0], vd.size()));
deque<double> d(vd.begin(), vd.end());                    // copy data into
                                                          // deque
list<double> l(vd.begin(), vd.end());                     // copy data into list
set<double> s(vd.begin(), vd.end());                      // copy data into set
```

```
void doSomething(const int* pInts, size_t numInts);   // C API (from above)
set<int> intSet;                                        // set that will hold
...                                                     // data to pass to API
vector<int> v(intSet.begin(), intSet.end());            // copy set data into
                                                        // a vector
if (!v.empty()) doSomething(&v[0], v.size());           // pass the data to
                                                        // the API
```

```
string s;
...                                   // make s large, then erase most
                                      // of its characters
string(s.begin(), s.end()).swap(s);   // do a “shrink-to-fit” on s
```

```
vector<Contestant> v;
string s;
...                                     // use v and s
vector<Contestant>().swap(v);           // clear v and minimize its capacity
string().swap(s);                       // clear s and minimize its capacity
```

```
T *p = &c[0];                       // initialize a T* with the address
                                    // of whatever operator[] returns
```

```
vector<bool> v;
bool *pb = &v[0];                    // initialize a bool* with the address of
                                     // what vector<bool>::operator[] returns
```

```
template <typename Allocator>
vector<bool, Allocator> {
public:
   class reference { ... };                  // class to generate proxies for
                                             // references to individual bits
   reference operator[](size_type n);        // operator[] returns a proxy
   ...
};
```

```
vector<bool> v;
bool *pb = &v[0];                      // error! the expression on the right is
                                       // of type vector<bool>::reference*,
                                       // not bool*
```

```
bool operator==(const Widget& lhs, const Widget& rhs)
{
   // code that ignores the lastAccessed field
}
```

```
!(w1 < w2)                    // it's not true that w1 < w2
&&                            // and
!(w2 < w1)                    // it's not true that w2 < w1
```

```
!c.key_comp()(x, y) && !c.key_comp()(y, x)    // it's not true that x precedes
                                              // y in c's sort order and it's
                                              // also not true that y precedes
                                              // x in c's sort order
```

```
struct CIStringCompare:                        // class for case-insensitive
  public                                       // string comparisons;
  binary_function<string, string, bool> {      // see Item 40 for info on
                                               // this base class
  bool operator()(const string& lhs,
                  const string& rhs) const
  {
    return ciStringCompare(lhs, rhs);          // see Item 35 for how
  }                                            // ciStringCompare is
                                               // implemented
};
```

```
set<string, CIStringCompare> ciss;                // ciss = “case-insensitive
                                                  //           string set”
```

```
ciss.insert("Persephone");          // a new element is added to the set
ciss.insert("persephone");          // no new element is added to the set
```

```
if (find( ciss.begin(), ciss.end(),
          "persephone") != ciss.end()) ...             // this test will fail
```

```
set<string*> ssp;                              // ssp = “set of string ptrs”

ssp.insert(new string("Anteater"));
ssp.insert(new string("Wombat"));
ssp.insert(new string("Lemur"));
ssp.insert(new string("Penguin"));
```

```
for (set<string*>::const_iterator i = ssp.begin();        // you expect to see
      i != ssp.end();                                       // this: “Anteater”,
      ++i)                                                  // “Lemur”, “Penguin”,
   cout << *i << endl;                                      // “Wombat”
```

```
copy(ssp.begin(), ssp.end(),                         // copy the strings in
     ostream_iterator<string>(cout, "\n"));          // ssp to cout (but this
                                                     // won’t compile)
```

```
struct StringPtrLess:
  public binary_function<const string*,             // see Item 40 for the
                         const string*,             // reason for this base
                         bool> {                    // class

  bool operator()(const string *ps1, const string *ps2) const
  {
    return *ps1 < *ps2;
  }

};
```

```
typedef set<string*, StringPtrLess> StringPtrSet;
StringPtrSet ssp;                             // create a set of strings and order
                                              // them as defined by StringPtrLess
...                                           // insert the same four strings as
                                              // before
```

```
for (StringPtrSet::const_iterator i = ssp.begin();       // prints “Anteater”,
       i != ssp.end();                                    // “Lemur”,
       ++i)                                               // “Penguin”,
   cout << **i << endl;                                   // “Wombat”
```

```
void print(const string *ps)                  // print to cout the
{                                             // object pointed to
   cout << *ps << endl;                       // by ps
}

for_each(ssp.begin(), ssp.end(), print);      // invoke print on each
                                              // element in ssp
// when functors of this type are passed a T*, they return a const T&
struct Dereference {

   template <typename T>
   const T& operator()(const T *ptr) const
   {
      return *ptr;
   }

};

transform( ssp.begin(), ssp.end(),                    // "transform" each
           ostream_iterator<string>(cout, "\n"),      // element in ssp by
           Dereference());                            // dereferencing it,
                                                      // writing the results
                                                      // to cout
```

```
bool stringPtrLess(const string* ps1,            // would-be comparison
                   const string* ps2)            // function for string*
{                                                // pointers to be sorted by
  return *ps1 < *ps2;                            // string value
}

set<string*, stringPtrLess> ssp;                 // attempt to use stringPtrLess
                                                 // as ssp's comparison function;
                                                 // this won't compile
```

```
struct DereferenceLess {                   // see Item 7 for why we're using
                                           // a class containing a templatized
                                           // operator()
   template <typename PtrType>
   bool operator()(PtrType pT1,            // parameters are passed by
                   PtrType pT2) const      // value, because we expect them
   {                                       // to be (or to act like) pointers
      return *pT1 < *pT2;
   }
};
```

```
set<string*, DereferenceLess> ssp;          // behaves the same as
                                            // set<string*, StringPtrLess>
```

```
struct StringPtrGreater:                          // highlights show how
  public binary_function<const string*,           // this code was changed
                         const string*,           // from page 89. Beware,
                         bool> {                  // this code is flawed!

  bool operator()(const string *ps1, const string *ps2) const
  {
    return !( *ps1 < *ps2);                       // just negate the old test;
  }                                               // this is incorrect!

};
```

```
struct StringPtrGreater:                              // this is a valid
  public binary_function<const string*,               // comparison type for
                         const string*,               // associative containers
                         bool> {

  bool operator()(const string *ps1, const string *ps2) const
  {
    return *ps2 < *ps1;                               // return whether *ps2
  }                                                   // precedes *ps1 (i.e., swap
                                                      // the order of the
};                                                    // operands)
```

```
multiset<int, less_equal<int> > s;           // s is still sorted by “<=”
s.insert(10);                                // insert 10_A
s.insert(10);                                // insert 10_B
```

```
map<int, string> m;
...
m.begin()->first = 10;                    // error! map keys can't be changed
multimap<int, string> mm;
...
mm.begin()->first = 20;                   // error! multimap keys can't
                                          // be changed, either
```

```
class Employee {
public:
   ...
   const string& name() const;                // get employee name
   void setName(const string& name);          // set employee name

   const string& title() const;               // get employee title
   void setTitle(const string& title);        // set employee title

   int idNumber() const;                      // get employee ID number
   ...
};
```

```
struct IDNumberLess:
   public binary_function<Employee, Employee, bool> {        // see Item 40
   bool operator()( const Employee& lhs,
                        const Employee& rhs) const
   {
      return lhs.idNumber() < rhs.idNumber();
   }
};
typedef set<Employee, IDNumberLess> EmpIDSet;
EmpIDSet se;                                                 // se is a set of employees
                                                             // sorted by ID number
```

```
Employee selectedID;                          // variable to hold employee
                                              // with selected ID number
...
EmpIDSet::iterator i = se.find(selectedID);
if (i != se.end()) {
   i->setTitle("Corporate Deity");            // give employee new title;
}                                             // this shouldn't compile —
                                              // see next page for why
```

```
EmpIDSet se;                          // as before, se is a set of employees
                                      // sorted by ID number
Employee selectedID;                  // as before, selectedID is a dummy
                                      // employee with the selected ID
                                      // number
...
EmpIDSet::iterator i = se.find(selectedID);
if (i != se.end()) {
   i->setTitle("Corporate Deity");    // some STL implementations
}                                     // accept this line, others reject it
```

```
EmpIDSet::iterator i = se.find(selectedID);
if (i != se.end()) {
   i->setTitle("Corporate Deity");          // some STL implementations will
}                                           // reject this line because *i is const
```

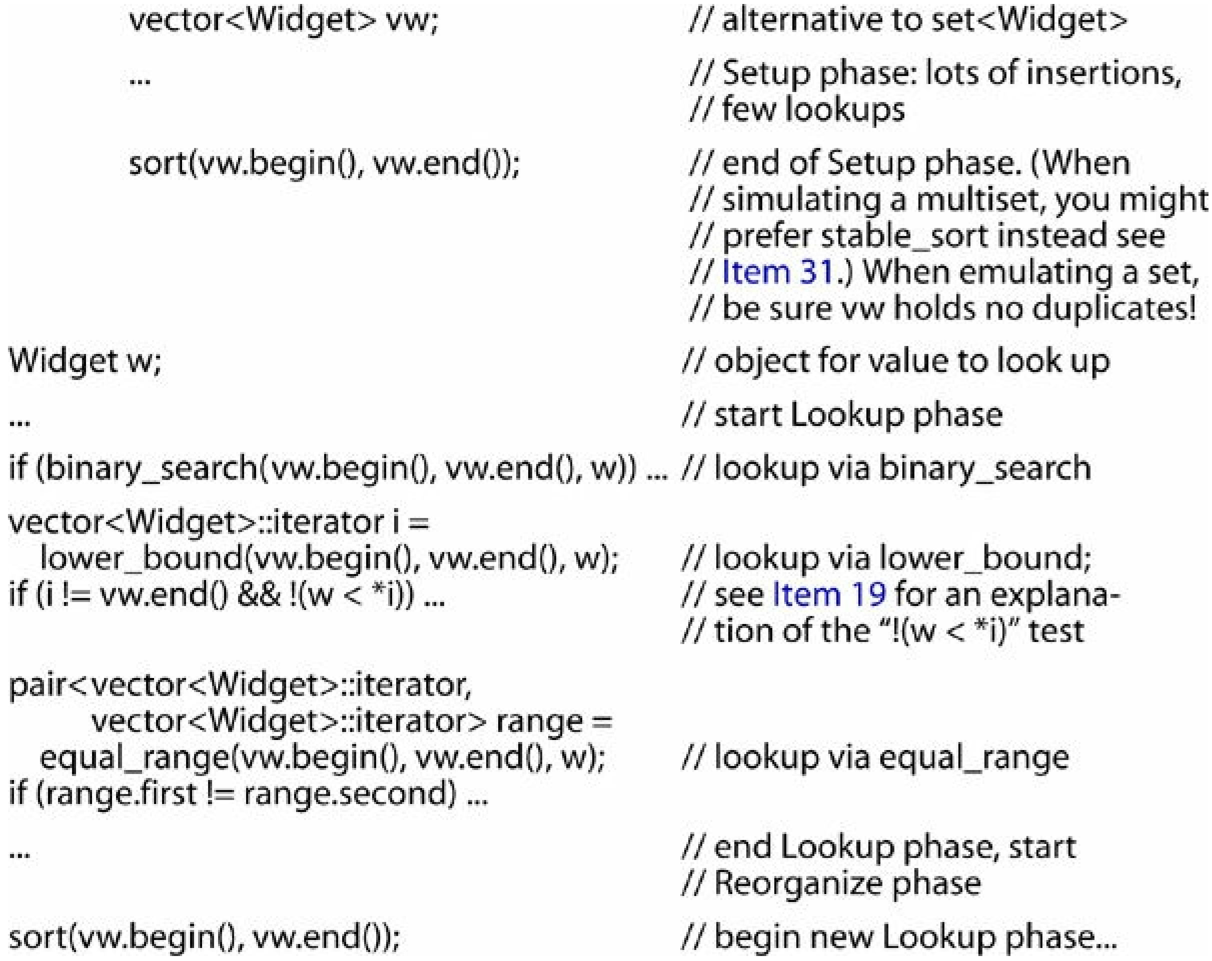
```
if (i != se.end()) {                                            // cast away
  const_cast<Employee&>(*i).setTitle("Corporate Deity");        // constness
}                                                               // of *i
```

```
if (i != se.end()) {                                                    // cast *i
   static_cast<Employee>(*i).setTitle("Corporate Deity");               // to an
}                                                                       // Employee
```

```
if (i != se.end()) {                                          // same as above,
   ((Employee)(*i)).setTitle("Corporate Deity");              // but using C
}                                                             // cast syntax
```

```
if (i != se.end()) {
   Employee tempCopy(*i);                        // copy *i into tempCopy
   tempCopy.setTitle("Corporate Deity");         // modify tempCopy
}
```

```
EmpIDSet se;                          // as before, se is a set of employees
                                      // sorted by ID number
Employee selectedID;                  // as before, selectedID is a dummy
                                      // employee with the desired ID number
...
EmpIDSet::iterator i =
  se.find(selectedID);                // Step 1: find element to change
if (i != se.end()) {
  Employee e(*i);                     // Step 2: copy the element
  e.setTitle("Corporate Deity");      // Step 3: modify the copy
  se.erase(i++);                      // Step 4: remove the element;
                                      // increment the iterator to maintain
                                      // its validity (see Item 9)
  se.insert(i, e);                    // Step 5: insert new value; hint that its
                                      // location is the same as that of the
}                                     // original element
```



```
    vector<Widget> vw;                       // alternative to set<Widget>
    ...                                      // Setup phase: lots of insertions,
                                             // few lookups
    sort(vw.begin(), vw.end());              // end of Setup phase. (When
                                             // simulating a multiset, you might
                                             // prefer stable_sort instead see
                                             // Item 31.) When emulating a set,
                                             // be sure vw holds no duplicates!
Widget w;                                    // object for value to look up
...                                          // start Lookup phase
if (binary_search(vw.begin(), vw.end(), w)) ... // lookup via binary_search
vector<Widget>::iterator i =
  lower_bound(vw.begin(), vw.end(), w);      // lookup via lower_bound;
if (i != vw.end() && !(w < *i)) ...          // see Item 19 for an explana-
                                             // tion of the "!(w < *i)" test
pair<vector<Widget>::iterator,
      vector<Widget>::iterator> range =
  equal_range(vw.begin(), vw.end(), w);      // lookup via equal_range
if (range.first != range.second) ...
...                                          // end Lookup phase, start
                                             // Reorganize phase
sort(vw.begin(), vw.end());                  // begin new Lookup phase...
```

```
typedef pair<string, int> Data;                // type held in the "map"
                                               // in this example

class DataCompare {                            // class for comparison
public:                                        // functions

  bool operator()(const Data& lhs,                 // comparison func
                  const Data& rhs) const           // for sorting
  {
    return keyLess(lhs.first, rhs.first);          // keyLess is below
  }

  bool operator()(const Data& lhs,                 // comparison func
                  const Data::first_type& k) const // for lookups
  {                                                // (form 1)
    return keyLess(lhs.first, k);
  }

  bool operator()( const Data::first_type& k,      // comparison func
                   const Data& rhs) const          // for lookups
  {                                                // (form 2)
    return keyLess(k, rhs.first);
  }

private:
  bool keyLess(const Data::first_type& k1,         // the "real"
               const Data::first_type& k2) const   // comparison
  {                                                // function
    return k1 < k2;
  }

};
```

```
vector<Data> vd;                              // alternative to
                                              // map<string, int>
...                                           // Setup phase: lots of
                                              // insertions, few lookups
sort(vd.begin(), vd.end(), DataCompare());    // end of Setup phase. Simu-
                                              // lated maps must avoid
                                              // dupes; simulated multimaps
                                              // might use stable_sort
string s;                                     // object for value to look up
...                                           // start Lookup phase
if (binary_search(vd.begin(), vd.end(), s,
                  DataCompare())) ...         // lookup via binary_search
vector<Data>::iterator i =
  lower_bound(vd.begin(), vd.end(), s,
              DataCompare());                 // lookup via lower_bound;
if (i != vd.end() && !DataCompare()(s, *i))... // again, see Item 19 for info
                                              // on the
                                              // "!DataCompare()(s, *i)" test
pair<vector<Data>::iterator,
     vector<Data>::iterator> range =
  equal_range(vd.begin(), vd.end(), s,
              DataCompare());                 // lookup via equal_range
if (range.first != range.second) ...
...                                           // end Lookup phase, start
                                              // Reorganize phase
sort(vd.begin(), vd.end(), DataCompare());    // begin new Lookup phase...
```

```
class Widget {
public:
   Widget();
   Widget(double weight);

   Widget& operator=(double weight);
   ...
};
```

```
typedef map<int, Widget> IntWidgetMap;                // convenience
                                                      // typedef

pair<IntWidgetMap::iterator, bool> result =           // create new map
   m.insert(IntWidgetMap::value_type(1, Widget()));   // entry with key 1
                                                      // and a default-
                                                      // constructed value
                                                      // object; see below
                                                      // for a comment on
                                                      // value_type

result.first->second = 1.50;                          // assign to the
                                                      // newly-constructed
                                                      // value object
```

```
m[k] = v;                                                // use operator[]
                                                         // to update k's
                                                         // value to be v
m.insert(IntWidgetMap::value_type(k, v)).first->second = v; // use insert to
                                                         // update k's
                                                         // value to be v
```

```
iterator affectedPair =                    // if key k isn't in map m, efficiently
  efficientAddOrUpdate(m, k, v);           // add pair (k,v) to m; otherwise
                                           // efficiently update to v the value
                                           // associated with k. Return an
                                           // iterator to the added or
                                           // modified pair
```

```
template< typename MapType,                           // type of map
          typename KeyArgType,                        // see below for why
          typename ValueArgType>                      // KeyArgType and
                                                      // ValueArgType are type
typename MapType::iterator                            // parameters
efficientAddOrUpdate(MapType& m,
                     const KeyArgType& k,
                     const ValueArgType& v)
{
  typename MapType::iterator lb =                     // find where k is or should
    m.lower_bound(k);                                 // be; see page 7B for why
                                                      // "typename" is needed
                                                      // here

  if (lb != m.end() &&                                // if lb points to a pair
     !(m.key_comp()(k, lb->first))) {                 // whose key is equiv to k...

    lb->second = v;                                   // update the pair's value
    return lb;                                        // and return an iterator
  }                                                   // to that pair

  else {
    typedef typename MapType::value_type MVT;
    return m.insert(lb, MVT(k, v));                   // add pair(k, v) to m and
  }                                                   // return an iterator to the
}                                                     // new map element
```

```
class Widget {                                    // also as before
public:
  ...
  Widget& operator=(double weight);
  ...
};
```

```
template<typename T,
         typename HashFunction,
         typename CompareFunction,
         typename Allocator = allocator<T> >
class hash_container;
```

```
template<typename T,
                typename HashFunction = hash<T>,
                typename CompareFunction = equal_to<T>,
                typename Allocator = allocator<T> >
class hash_set;
```

```
template<typename T, typename CompareFunction>
class hash_compare;

template<typename T,
            typename HashingInfo = hash_compare<T, less<T> >,
            typename Allocator = allocator<T> >
class hash_set;
```

```
template<typename T, typename CompareFunction = less<T> >
class hash_compare {
public:

   enum {
      bucket_size = 4,                    // max ratio of elements to buckets
      min_buckets = 8                     // minimum number of buckets

   };

   size_t operator()(const T&) const;     // hash function

   bool operator()(const T&,              // comparison function
                   const T&) const;

   ...                                    // a few things omitted, including
};                                        // the use of CompareFunction
```

```
iterator insert(iterator position, const T& x);
iterator erase(iterator position);
iterator erase(iterator rangeBegin, iterator rangeEnd);
```

```
typedef deque<int> IntDeque;                      // STL container and
typedef IntDeque::iterator Iter;                  // iterator types are easier
typedef IntDeque::const_iterator ConstIter;       // to work with if you
                                                  // use some typedefs
Iter i;
ConstIter ci;
...                                               // make i and ci point into
                                                  // the same container
if (i == ci) ...                                  // compare an iterator
                                                  // and a const_iterator
```

```
if (ci == i) ...                            // workaround for when the
                                            // above won’t compile
```

```
if (static_cast<ConstIter>(i) - ci >= 3) ...      // workaround for when the
                                                  // above won't compile
```

```
typedef deque<int> IntDeque;                    // convenience typedefs
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;
ConstIter ci;                                   // ci is a const_iterator
...
Iter i(ci);                               // error! no implicit conversion from
                                          // const_iterator to iterator
Iter i(const_cast<Iter>(ci));             // still an error! can't cast a
                                          // const_iterator to an iterator
```

```
typedef deque<int> IntDeque;                    // as before
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;

IntDeque d;

ConstIter ci;
...                                     // make ci point into d

Iter i(d.begin());                      // initialize i to d.begin()

advance(i, distance(i, ci));            // move i up to where ci is
                                        // (but see below for why this must
                                        // be tweaked before it will compile)
```

```
template <typename InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

```
advance(i, distance<ConstIter>(i, ci));        // figure the distance between
                                               // i and ci (as const_iterators),
                                               // then move i that distance
```

```
vector<int> v;
v.reserve(5);                              // see Item 14

for (int i = 1; i <= 5; ++i) {             // put 1-5 in the vector
  v.push_back(i);
}

vector<int>::reverse_iterator ri =         // make ri point to the 3
  find(v.rbegin(), v.rend(), 3);

vector<int>::iterator i(ri.base());        // make i the same as ri's base
```

```
vector<int> v;
...                                            // as above, put 1-5 in v
vector<int>::reverse_iterator ri =             // as above, make ri point to the 3
  find(v.rbegin(), v.rend(), 3);
v.erase(--ri.base());                          // attempt to erase at the position
                                               // preceding ri.base(); for a vector,
                                               // this will typically not compile
```

```
...                                  // as above
v.erase((++ri).base());              // erase the element pointed to by
                                     // ri; this should always compile
```

```
ifstream inputFile("interestingData.txt");
string fileData( (istream_iterator<char>(inputFile)),   // read inputFile into
                  istream_iterator<char>());             // fileData; see below
                                                         // for why this isn't
                                                         // quite right, and see
                                                         // Item 6 for a warning
                                                         // about this syntax
```

```
ifstream inputFile("interestingData.txt");
inputFile.unsetf(ios::skipws);                    // disable the skipping of
                                                  // whitespace in inputFile
string fileData((istream_iterator<char>(inputFile)),
                istream_iterator<char>());
```

```
ifstream inputFile("interestingData.txt");
string fileData((istreambuf_iterator<char>(inputFile)),
                 istreambuf_iterator<char>());
```

```
int transmogrify(int x);                      // this function produces
                                              // some new value from x

vector<int> values;

...                                           // put data into values

vector<int> results;                          // apply transmogrify to
transform( values.begin(), values.end(),      // each object in values,
           results.end(),                     // appending the return
           transmogrify);                     // values to results; this
                                              // code has a bug!
```

```
vector<int> results;                          // apply transmogrify to
transform( values.begin(), values.end(),      // each object in values,
           back_inserter(results),            // inserting the return
           transmogrify);                     // values at the end of
                                              // results
```

```
...                                              // same as before
list<int> results;                               // results is now a list
transform( values.begin(), values.end(),         // insert transform's
           front_inserter(results),              // results at the front of
           transmogrify);                        // results in reverse order
```

```
list<int> results;                                  // same as before

transform( values.rbegin(), values.rend(),          // insert transform's
           front_inserter(results),                 // results at the front of
           transmogrify);                           // results; preserve the
                                                    // relative object ordering
```

```
vector<int> values;                                   // as before
...
vector<int> results;                                  // as before, except now
...                                                   // results has some data
                                                      // in it prior to the call to
                                                      // transform
transform( values.begin(), values.end(),                          // insert the
           inserter(results, results.begin() + results.size() / 2), // results of
           transmogrify);                                         // the trans-
                                                                  // mogrifica-
                                                                  // tions at
                                                                  // the middle
                                                                  // of results
```

```
vector<int> values;                                // as above
vector<int> results;
...
results.reserve(results.size() + values.size());   // ensure that results has
                                                   // the capacity for at least
                                                   // values.size() more
                                                   // elements
transform( values.begin(), values.end(),                                // as above,
           inserter(results, results.begin() + results.size() / 2),     // but results
           transmogrify);                                               // won't do
                                                                        // any reallo-
                                                                        // cations
```

```
vector<int> values;                                  // as above
vector<int> results;
...
results.reserve(results.size() + values.size());     // as above
transform( values.begin(), values.end(),             // write the results of
           results.end(),                            // the transmogrifications
           transmogrify);                            // to uninitialized memory;
                                                     // behavior is undefined!
```

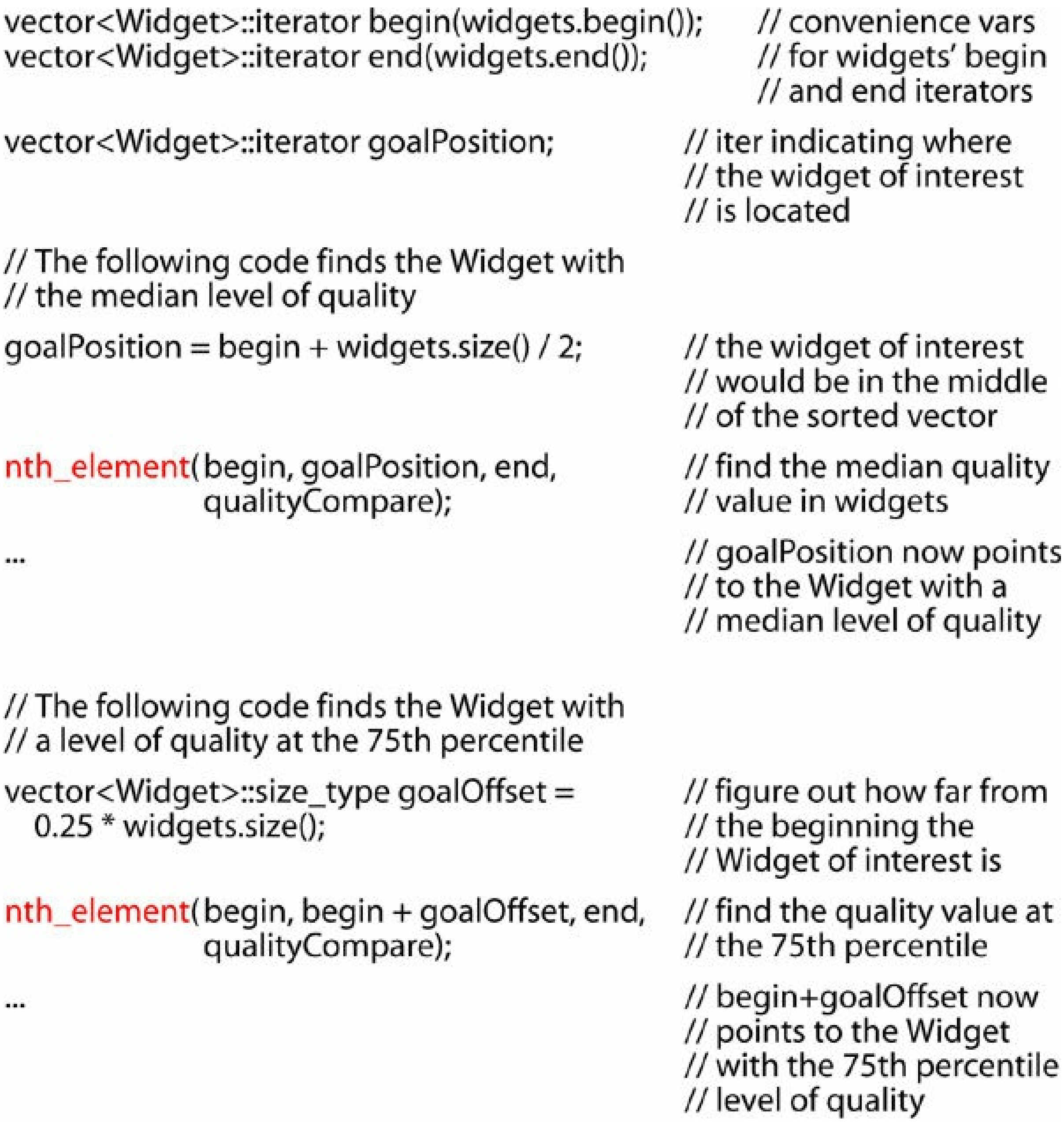
```
vector<int> values;                              // as above
vector<int> results;
...
results.reserve(results.size() + values.size());  // as above

transform( values.begin(), values.end(),         // write the results of
           back_inserter(results),               // the transmogrifications
           transmogrify);                        // to the end of results,
                                                 // avoiding reallocations
                                                 // during the process
```

```
vector<int> values;
vector<int> results;
...
if (results.size() < values.size()) {           // make sure results is at
   results.resize(values.size());               // least as big as values is
}
transform(values.begin(), values.end(),         // overwrite the first
          results.begin(),                      // values.size() elements of
          transmogrify);                        // results
```

```
...
results.clear();                                  // destroy all elements in
                                                  // results
results.reserve(values.size());                   // reserve enough space
transform( values.begin(), values.end(),          // put transform's return
           back_inserter(results),                // values into results
           transmogrify);
```

```
bool qualityCompare(const Widget& lhs, const Widget& rhs)
{
   // return whether lhs's quality is greater than rhs's quality
}

...

partial_sort(widgets.begin(),                   // put the best 20 elements
             widgets.begin() + 20,              // (in order) at the front of
             widgets.end(),                     // widgets
             qualityCompare);

...                                             // use widgets...
```

```
nth_element(widgets.begin(),              // put the best 20 elements
            widgets.begin() + 19,         // at the front of widgets,
            widgets.end(),                // but don't worry about
            qualityCompare);              // their order
```



```
vector<Widget>::iterator begin(widgets.begin());     // convenience vars
vector<Widget>::iterator end(widgets.end());         // for widgets' begin
                                                     // and end iterators

vector<Widget>::iterator goalPosition;           // iter indicating where
                                                 // the widget of interest
                                                 // is located

// The following code finds the Widget with
// the median level of quality

goalPosition = begin + widgets.size() / 2;       // the widget of interest
                                                 // would be in the middle
                                                 // of the sorted vector

nth_element(begin, goalPosition, end,            // find the median quality
            qualityCompare);                     // value in widgets

...                                              // goalPosition now points
                                                 // to the Widget with a
                                                 // median level of quality


// The following code finds the Widget with
// a level of quality at the 75th percentile

vector<Widget>::size_type goalOffset =           // figure out how far from
   0.25 * widgets.size();                        // the beginning the
                                                 // Widget of interest is

nth_element(begin, begin + goalOffset, end,      // find the quality value at
            qualityCompare);                     // the 75th percentile

...                                              // begin+goalOffset now
                                                 // points to the Widget
                                                 // with the 75th percentile
                                                 // level of quality
```

```
bool hasAcceptableQuality(const Widget& w)
{
   // return whether w has a quality rating of 2 or better;
}
```

```
vector<Widget>::iterator goodEnd =          // move all widgets satisfying
   partition( widgets.begin(),              // hasAcceptableQuality to
              widgets.end(),                // the front of widgets, and
              hasAcceptableQuality);        // return an iterator to the first
                                            // widget that isn't satisfactory
```

```
vector<int> v;                       // create a vector<int> and fill it with
v.reserve(10);                       // the values 1-10. (See Item 14 for an
for (int i = 1; i <= 10; ++i) {      // explanation of the reserve call.)
   v.push_back(i);
}

cout << v.size();                    // prints 10

v[3] = v[5] = v[9] = 99;             // set 3 elements to 99

remove(v.begin(), v.end(), 99);      // remove all elements with value 99

cout << v.size();                    // still prints 10!
```

```
vector<int> v;                                      // as before
...
v.erase(remove(v.begin(), v.end(), 99), v.end());   // really remove all
                                                    // elements with value 99
cout << v.size();                                   // now returns 7
```

```
list<int> li;                       // create a list
...                                 // put some values into it
li.remove(99);                      // eliminate all elements with value 99;
                                    // this really removes elements, so li’s
                                    // size may change
```

```
class Widget {
public:
   ...
   bool isCertified() const;                // whether the Widget is certified
   ...
};

vector<Widget*> v;                          // create a vector and fill it with
...                                         // pointers to dynamically
v.push_back(new Widget);                    // allocated Widgets
...
```

```
v.erase( remove_if(v.begin(), v.end(),                    // erase ptrs to
                   not1(mem_fun(&Widget::isCertified))),  // uncertified
     v.end());                                            // Widgets; see
                                                          // Item 41 for
                                                          // info on
                                                          // mem_fun
```

```
void delAndNullifyUncertified(Widget*& pWidget)    // if *pWidget is an
{                                                  // uncertified Widget,
  if (!pWidget->isCertified()) {                   // delete the pointer
    delete pWidget;                                // and set it to null
    pWidget = 0;
  }

}

for_each( v.begin(), v.end(),                      // delete and set to
          delAndNullifyUncertified);               // null all ptrs to
                                                   // uncertified widgets

v.erase( remove(v.begin(), v.end(),                // eliminate null ptrs
                static_cast<Widget*>(0)),          // from v; 0 must be
        v.end());                                  // cast to a ptr so C++
                                                   // correctly deduces
                                                   // the type of
                                                   // remove's 3rd param
```

```
template<typename T>                    // RCSP = "Reference Counting
class RCSP { ... };                     //             Smart Pointer"

typedef RCSP<Widget> RCSPW;             // RCSPW = "RCSP to Widget"

vector<RCSPW> v;                        // create a vector and fill it with
...                                     // smart pointers to dynamically
v.push_back(RCSPW(new Widget));         // allocated Widgets
...

v.erase(remove_if(v.begin(), v.end(),                          // erase the ptrs
                  not1(mem_fun(&Widget::isCertified))),        // to uncertified
        v.end());                                              // Widgets; no
                                                               // resources are
                                                               // leaked
```

```
vector<int> v;                                  // create a vector, put some
...                                             // data into it, sort it into
sort(v.begin(), v.end(), greater<int>());       // descending order
...                                             // work with the vector
                                                // (without changing it)
bool a5Exists =                                 // search for a 5 in the vector,
  binary_search(v.begin(), v.end(), 5);         // assuming it's sorted in
                                                // ascending order!
```

```
bool a5Exists =                                              // search for a 5
   binary_search(v.begin(), v.end(), 5, greater<int>());     // using greater as
                                                             // the comparison
                                                             // function
```

```
int ciCharCompare(char c1, char c2)     // case-insensitively compare chars
{                                       // c1 and c2, returning -1 if c1 < c2,
                                        // 0 if c1==c2, and 1 if c1 > c2

  int lc1 = tolower(static_cast<unsigned char>(c1));      // see below for
  int lc2 = tolower(static_cast<unsigned char>(c2));      // info on these
                                                          // statements

  if (lc1 < lc2) return -1;

  if (lc1 > lc2) return 1;

  return 0;
}
```

```
int ciStringCompareImpl(const string& s1,              // see below for
                        const string& s2);              // implementation

int ciStringCompare(const string& s1, const string& s2)
{
  if (s1.size() <= s2.size()) return ciStringCompareImpl(s1, s2);
  else return -ciStringCompareImpl(s2, s1);
}
```

```
int ciStringCompareImpl(const string& s1, const string& s2)
{
   typedef pair<string::const_iterator,                  // PSCI = "pair of
                    string::const_iterator> PSCI;         // string::const_iterator"

    PSCI p = mismatch(                                    // see below for an
                s1.begin(), s1.end(),                     // explanation of why
                s2.begin(),                               // we need not2; see
                not2(ptr_fun(ciCharCompare)));            // Item 41 for why we
                                                          // need ptr_fun

   if (p.first == s1.end()) {                             // if true, either s1 and
      if (p.second == s2.end()) return 0;                 // s2 are equal or
      else return -1;                                     // s1 is shorter than s2
   }

   return ciCharCompare(*p.first, *p.second);             // the relationship of the
}                                                         // strings is the same as
                                                          // that of the
                                                          // mismatched chars
```

```
bool ciCharLess(char c1, char c2)                  // return whether c1
{                                                  // precedes c2 in a case-
   return                                          // insensitive comparison;
      tolower(static_cast<unsigned char>(c1)) <    // Item 46 explains why a
      tolower(static_cast<unsigned char>(c2));     // function object might
}                                                  // be preferable to this
                                                   // function

bool ciStringCompare(const string& s1, const string& s2)
{
   return lexicographical_compare(s1.begin(), s1.end(),     // see below for
                                  s2.begin(), s2.end(),     // a discussion of
                                  ciCharLess);              // this algorithm
}                                                           // call
```

```
int ciStringCompare(const string& s1, const string& s2)
{
   return stricmp(s1.c_str(), s2.c_str());          // the function name on
}                                                   // your system might
                                                    // not be stricmp
```

copy
replace_copy
replace_copy_if
remove_copy
remove_copy_if
uninitialized_copy
copy_backward
reverse_copy
unique_copy
rotate_copy
partial_sort_copy

```
vector<Widget> widgets;
...
copy_if( widgets.begin(), widgets.end(),               // this won't compile;
         ostream_iterator<Widget>(cerr, "\n"),         // there is no copy_if
         isDefective);                                 // in the STL
```

```
template< typename InputIterator,                     // a not-quite-right
          typename OutputIterator,                    // implementation of
          typename Predicate>                         // copy_if
OutputIterator copy_if( InputIterator begin,
                        InputIterator end,
                        OutputIterator destBegin,
                        Predicate p)
{
  return remove_copy_if(begin, end, destBegin, not1(p));
}
```

```
copy_if( widgets.begin(), widgets.end(),            // well-intentioned code
         ostream_iterator<Widget>(cerr, "\n"),      // that will not compile
         isDefective);
```

```
template< typename InputIterator,                    // a correct
          typename OutputIterator,                   // implementation of
          typename Predicate>                        // copy_if
OutputIterator copy_if( InputIterator begin,
                        InputIterator end,
                        OutputIterator destBegin,
                        Predicate p)
{
  while (begin != end) {
    if (p(*begin)) *destBegin++ = *begin;
    ++begin;
  }

  return destBegin;
}
```

```
list<double> ld;                                          // create a list and put
...                                                       // some doubles into it
double sum = accumulate(ld.begin(), ld.end(), 0.0);       // calculate their sum,
                                                          // starting at 0.0
```

```
double sum = accumulate(ld.begin(), ld.end(), 0);     // calculate their sum,
                                                      // starting at 0; this
                                                      // is not correct!
```

```
cout << "The sum of the ints on the standard input is "   // print the sum of
     << accumulate(istream_iterator<int>(cin),            // the ints in cin
                   istream_iterator<int>(),
                   0);
```

```
string::size_type                                      // see below for info
stringLengthSum(string::size_type sumSoFar,            // on string::size_type
                const string& s)
{
   return sumSoFar + s.size();
}
```

```
set<string> ss;                                  // create container of strings,
...                                              // and populate it
string::size_type lengthSum =                    // set lengthSum to the result
  accumulate(ss.begin(), ss.end(),               // of calling stringLengthSum on
    static_cast<string::size_type>(0),           // each element in ss, using 0
    stringLengthSum);                            // as the initial summary value
```

```
vector<float> vf;                           // create container of floats
...                                         // and populate it
float product =                             // set product to the result of
   accumulate(vf.begin(), vf.end(),         // calling multiplies<float> on
              1.0f, multiplies<float>());   // each element in vf, using 1.0f
                                            // as the initial summary value
```

```
struct Point {
   Point(double initX, double initY): x(initX), y(initY) {}
   double x, y;
};
```

```
list<Point> lp;
...
Point avg =                                   // average the points in lp
   accumulate( lp.begin(), lp.end(),
               Point(0, 0), PointAverage());
```

```
class PointAverage:
   public binary_function<Point, Point, Point> {                    // see Item 40
public:
   PointAverage(): numPoints(0), xSum(0), ySum(0) {}

   const Point operator()(const Point& avgSoFar, const Point& p)
   {
      ++numPoints;
      xSum += p.x;
      ySum += p.y;

      return Point(xSum/numPoints, ySum/numPoints);
   }

private:
   size_t numPoints;
   double xSum;
   double ySum;
};
```

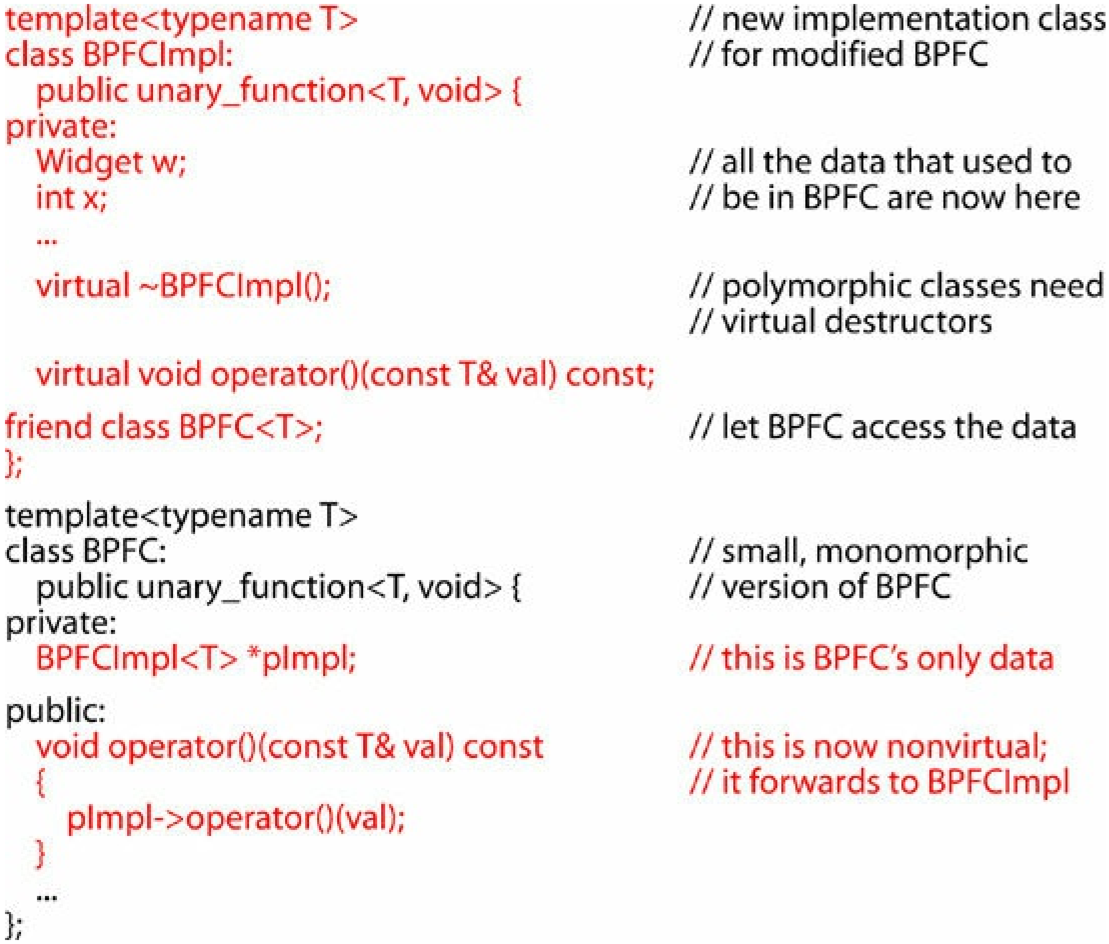
```
struct Point { ... };                                         // as before

class PointAverage:
  public unary_function<Point, void> {                         // see Item 40
public:
  PointAverage(): numPoints(0), xSum(0), ySum(0) {}

  void operator()(const Point& p)
  {
    ++numPoints;
    xSum += p.x;
    ySum += p.y;
  }

  Point result() const
  {
    return Point(xSum/numPoints, ySum/numPoints);
  }

private:
  size_t numPoints;
  double xSum;
  double ySum;
};

list<Point> lp;
...
Point avg = for_each(lp.begin(), lp.end(), PointAverage()).result();
```

```
void qsort( void *base, size_t nmemb, size_t size,
                int (*cmpfcn)(const void*, const void*));
```

```
template<class InputIterator,
         class Function>
Function                                // note return-by-value
for_each( InputIterator first,
          InputIterator last,
          Function f);                  // note pass-by-value
```

```
class DoSomething:
   public unary_function<int, void> {          // Item 40 explains base class
public:
   void operator()(int x) { ... }

   ...

};

typedef deque<int>::iterator DequeIntIter;   // convenience typedef

deque<int> di;
...
DoSomething d;                               // create a function object
...
for_each<DequeIntIter,                       // call for_each with type
              DoSomething&>(di.begin(),      // parameters of DequeIntIter
                            di.end(),        // and DoSomething&;
                            d);              // this forces d to be
                                             // passed and returned
                                             // by reference
```

```
template<typename T>
class BPFC:                                   // BPFC = " Big Polymorphic
  public                                      //            Functor Class"
    unary_function<T, void> {                 // Item 40 explains this
                                              // base class

private:
  Widget w;                                   // this class has lots of data,
  int x;                                      // so it would be inefficient
  ...                                         // to pass it by value

public:
   virtual void operator()(const T& val) const;  // this is a virtual function,
  ...                                         // so slicing would be bad
};
```



```
template<typename T>                          // new implementation class
class BPFCImpl:                               // for modified BPFC
  public unary_function<T, void> {
private:
  Widget w;                                   // all the data that used to
  int x;                                      // be in BPFC are now here
  ...
  virtual ~BPFCImpl();                        // polymorphic classes need
                                              // virtual destructors

  virtual void operator()(const T& val) const;

friend class BPFC<T>;                         // let BPFC access the data
};

template<typename T>
class BPFC:                                   // small, monomorphic
  public unary_function<T, void> {            // version of BPFC
private:
  BPFCImpl<T> *pImpl;                         // this is BPFC's only data

public:
  void operator()(const T& val) const         // this is now nonvirtual;
  {                                           // it forwards to BPFCImpl
    pImpl->operator()(val);
  }
  ...
};
```

```
class BadPredicate:                                  // see Item 40 for info
   public unary_function<Widget, bool> {             // on the base class
public:
   BadPredicate(): timesCalled(0) {}                 // init timesCalled to 0

   bool operator()(const Widget&)
   {
      return ++timesCalled == 3;
   }

private:
   size_t timesCalled;
};
```

```
vector<Widget> vw;                              // create vector and put some
...                                             // Widgets into it

vw.erase(remove_if( vw.begin(),                 // eliminate the third Widget;
                    vw.end(),                   // see Item 32 for info on how
                    BadPredicate()),            // erase and remove_if relate
         vw.end());
```

```
template <typename FwdIterator, typename Predicate>
FwdIterator remove_if(FwdIterator begin, FwdIterator end, Predicate p)
{
   begin = find_if(begin, end, p);

   if (begin == end) return begin;
   else {
      FwdIterator next = begin;
      return remove_copy_if(++next, end, begin, p);
   }
}
```

```
class BadPredicate:
   public unary_function<Widget, bool> {
public:
   bool operator()(const Widget&) const
   {
      return ++timesCalled == 3;                // error! can't change local data
   }                                            // in a const member function

   ...

};
```

```
bool anotherBadPredicate(const Widget&, const Widget&)
{
   static int timesCalled = 0;            // No! No! No! No! No! No! No!
   return ++timesCalled == 3;             // Predicates should be pure functions,
}                                         // and pure functions have no state
```

```
list<Widget*>::iterator i = find_if(widgetPtrs.begin(), widgetPtrs.end(),
                                    isInteresting);
if (i != widgetPtrs.end()) {
  ...                                                   // process the first
}                                                       // ptr to an interesting
                                                        // Widget
```

```
list<Widget*>::iterator i =
   find_if(widgetPtrs.begin(), widgetPtrs.end(),
           not1(isInteresting));                    // error! won’t compile
```

```
list<Widget*>::iterator i =
   find_if(widgetPtrs.begin(), widgetPtrs.end(),
              not1(ptr_fun(isInteresting)));                    // fine

if (i != widgetPtrs.end()) {
   ...                                                          // process the first ptr
}                                                               // to a dull Widget
```

```
template<typename T>
class MeetsThreshold: public std::unary_function<Widget, bool> {
private:
   const T threshold;

public:
   MeetsThreshold(const T& threshold);

   bool operator()(const Widget&) const;

   ...

};

struct WidgetNameCompare:
   public std::binary_function<Widget, Widget, bool> {

   bool operator()(const Widget& lhs, const Widget& rhs) const;

};
```

```
struct WidgetNameCompare:
   public std::binary_function<Widget, Widget, bool> {
   bool operator()(const Widget& lhs, const Widget& rhs) const;
};
```

```
struct PtrWidgetNameCompare:
   public std::binary_function<const Widget*, const Widget*, bool> {
   bool operator()(const Widget* lhs, const Widget* rhs) const;
};
```

```
list<Widget> widgets;

...

list<Widget>::reverse_iterator i1 =                   // find the last widget
   find_if( widgets.rbegin(), widgets.rend(),         // that fails to meet the
            not1(MeetsThreshold<int>(10)));           // threshold of 10
                                                      // (whatever that means)

Widget w( constructor arguments );

list<Widget>::iterator i2 =                           // find the first widget
   find_if( widgets.begin(), widgets.end(),           // that precedes w in the
            bind2nd(WidgetNameCompare(), w));         // sort order defined by
                                                      // WidgetNameCompare
```

```
f(x);                     // Syntax #1: When f is a
                          // non-member function
x.f();                    // Syntax #2: When f is a member
                          // function and x is an object or
                          // a reference to an object
p->f();                   // Syntax #3: When f is a member
                          // function and p is a pointer to x
```

```
void test(Widget& w);                    // test w and mark it as "failed" if
                                         // it doesn't pass the test
```

```
class Widget {
public:
  ...
  void test();                    // perform a self-test; mark *this
  ...                             // as “failed” if it doesn’t pass
};
```

```
for_each( vw.begin(), vw.end(),
          &Widget::test);                    // Call #2 (won't compile)
```

```
list<Widget*> lpw;                      // lpw holds pointers to widgets
for_each( lpw.begin(), lpw.end(),
          &Widget::test);               // Call #3 (also won't compile)
```

```
template<typename InputIterator, typename Function>
Function for_each(InputIterator begin, InputIterator end, Function f)
{
   while (begin != end) f(*begin++);
}
```

```
template<typename R, typename C>       // declaration for mem_fun for
mem_fun_t<R,C>                         // non-const member functions
mem_fun(R (C::*pmf)());                // taking no parameters. C is the
                                       // class, R is the return type of the
                                       // pointed-to member function
```

```
list<Widget*> lpw;                              // same as above
...
for_each(lpw.begin(), lpw.end(),
         mem_fun(&Widget::test));               // this will now compile
```

```
for_each(vw.begin(), vw.end(), &Widget::test);        // as above, Call #2;
                                                      // doesn’t compile
for_each(lpw.begin(), lpw.end(), &Widget::test);      // as above, Call #3;
                                                      // doesn’t compile
```

```
for_each(vw.begin(), vw.end(), ptr_fun(test));        // compiles and behaves
                                                      // like Call #1 above
```

```
class Widget {
public:
    ...
    size_t weight() const;
    size_t maxSpeed() const;

    ...
};
```

```
bool operator<(const Widget& lhs, const Widget& rhs)
{
   return lhs.weight() < rhs.weight();
}
```

```
template<>                                       // This is a specialization
struct std::less<Widget>:                        // of std::less for Widget;
  public                                         // it's also a very bad idea
  std::binary_function<Widget,
                       Widget,                   // See Item 40 for info
                       bool> {                   // on this base class

  bool operator()(const Widget& lhs, const Widget& rhs) const
  {
    return lhs.maxSpeed() < rhs.maxSpeed();
  }

};
```

```
namespace std {

  template<typename T>                          // this is a spec. of std::less
  struct less< boost::shared_ptr<T> >:          // for boost::shared_ptr<T>
    public                                      // (boost is a namespace)
    binary_function<boost::shared_ptr<T>,
                    boost::shared_ptr<T>,       // this is the customary
                    bool> {                     // base class (see Item 40)

    bool operator()( const boost::shared_ptr<T>& a,
                    const boost::shared_ptr<T>& b) const
    {
      return less<T*>()(a.get(),b.get());       // shared_ptr::get returns
    }                                           // the built-in pointer that's
                                                // in the shared_ptr object

  };

}
```

```
struct MaxSpeedCompare:
   public binary_function<Widget, Widget, bool> {

   bool operator()(const Widget& lhs, const Widget& rhs) const
   {
      return lhs.maxSpeed() < rhs.maxSpeed();
   }

};
```

```
list<Widget> lw;
...
for (list<Widget>::iterator i = lw.begin(); i != lw.end(); ++i) {
   i->redraw();
}
```

```
for_each(lw.begin(), lw.end(),                    // see Item 41 for info
         mem_fun_ref(&Widget::redraw));           // on mem_fun_ref
```

```
for (list<Widget>::iterator i = lw.begin(); i != lw.end(); ++i) {
    i->redraw();
}
```

```
// C API: this function takes a pointer to an array of at most arraySize
// doubles and writes data to it. It returns the number of doubles written.
size_t fillArray(double *pArray, size_t arraySize);

double data[maxNumDoubles];                      // create local array of
                                                 // max possible size

deque<double> d;                                 // create deque, put
...                                              // data into it

size_t numDoubles =
  fillArray(data, maxNumDoubles);                // get array data from API

for (size_t i = 0; i < numDoubles; ++i) {        // for each i in data,
  d.insert(d.begin(), data[i] + 41);             // insert data[i]+41 at the
}                                                // front of d; this code
                                                 // has a bug!
```

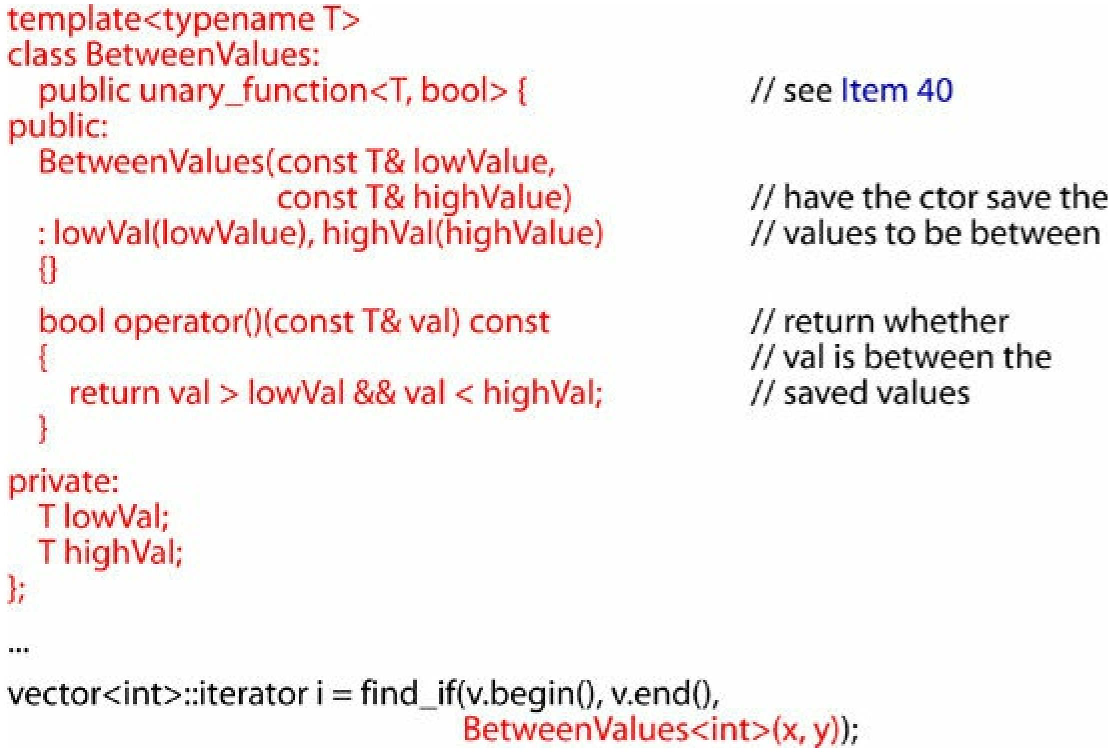
```
deque<double>::iterator insertLocation = d.begin();    // remember d's
                                                       // begin iterator

for (size_t i = 0; i < numDoubles; ++i) {              // insert data[i]+41
   d.insert(insertLocation++, data[i] + 41);           // at insertLocation, then
}                                                      // increment
                                                       // insertLocation; this
                                                       // code is also buggy!
```

```
deque<double>::iterator insertLocation =
   d.begin();                                  // as before

for (size_t i = 0; i < numDoubles; ++i) {      // update insertLocation
   insertLocation =                            // each time insert is
      d.insert(insertLocation, data[i] + 41);  // called to keep the
   ++insertLocation;                           // iterator valid, then
}                                              // increment it
```

```
transform(data, data + numDoubles,              // copy all elements
          inserter(d, d.begin()),               // from data to the front
          bind2nd(plus<double>(), 41));         // of d, adding 41 to each
```

```
vector<int> v;
int x, y;
...
vector<int>::iterator i = v.begin();      // iterate from v.begin() until an
for( ; i != v.end(); ++i) {               // appropriate value is found or
   if (*i > x && *i < y) break;           // v.end() is reached
}
...                                       // i now points to the value or is
                                          // the same as v.end()
```

```
vector<int>::iterator i =
  find_if( v.begin(), v.end(),                              // find the first value val
           compose2( logical_and<bool>(),                   // where the “and” of
                     bind2nd(greater<int>(), x),            // val > x and
                     bind2nd(less<int>(), y)));             // val < y
                                                            // is true
```



```
template<typename T>
class BetweenValues:
  public unary_function<T, bool> {                    // see Item 40
public:
  BetweenValues(const T& lowValue,
                const T& highValue)                   // have the ctor save the
  : lowVal(lowValue), highVal(highValue)              // values to be between
  {}

  bool operator()(const T& val) const                 // return whether
  {                                                   // val is between the
    return val > lowVal && val < highVal;             // saved values
  }

private:
  T lowVal;
  T highVal;
};

...

vector<int>::iterator i = find_if(v.begin(), v.end(),
                                  BetweenValues<int>(x, y));
```

```
{                                                   // beginning of function
    ...
    template <typename T>
    class BetweenValues: public unary_function<T, bool> { ... };

    vector<int>::iterator i = find_if(v.begin(), v.end(),
                                      BetweenValues<int>(x, y));

    ...
}                                                   // end of function
```

```
{                                                       // beginning of function
   ...
   class BetweenValues: public unary_function<int, bool> { ... };
   vector<int>::iterator i = find_if(v.begin(), v.end(),
                                     BetweenValues(x, y));

   ...
}                                                       // end of function
```

```
set<int> s;                                               // create set, put
...                                                       // 1,000,000 values
                                                          // into it

set<int>::iterator i = s.find(727);                       // use find member
if (i != s.end()) ...                                     // function

set<int>::iterator i = find(s.begin(), s.end(), 727);     // use find algorithm
if (i != s.end()) ...
```

```
list<Widget> lw;                        // list of Widgets
Widget w;                               // special Widget value
...
if (count(lw.begin(), lw.end(), w)) {
   ...                                  // w is in lw
} else {
   ...                                  // it's not
}
```

```
if (find(lw.begin(), lw.end(), w) != lw.end()) {
    ...
} else {
    ...
}
```

```
list<Widget>::iterator i = find(lw.begin(), lw.end(), w);
if (i != lw.end()) {
    ...                                          // found it, i points to the first one
} else {
    ...                                          // didn't find it
}
```

```
vector<Widget> vw;                            // create vector, put
...                                           // data into it, sort the
sort(vw.begin(), vw.end());                   // data
Widget w;                                     // value to search for
...
if (binary_search(vw.begin(), vw.end(), w)) {
   ...                                        // w is in vw
} else {
   ...                                        // it's not
}
```

```
vector<Widget>::iterator i = lower_bound(vw.begin(), vw.end(), w);
if (i != vw.end() && *i == w) {               // make sure i points to an object;
                                              // make sure the object has the
                                              // correct value; this has a bug!
   ...                                        // found the value, i points to the
                                              // first object with that value
} else {
   ...                                        // not found
}
```

```
vector<Widget> vw;
...
sort(vw.begin(), vw.end());

typedef vector<Widget>::iterator VWIter;    // convenience typedefs
typedef pair<VWIter, VWIter> VWIterPair;

VWIterPair p = equal_range(vw.begin(), vw.end(), w);

if (p.first != p.second) {                   // if equal_range didn't return
                                             // an empty range...

   ...                                       // found it, p.first points to the
                                             // first one and p.second
                                             // points to one past the last
} else {

   ...                                       // not found, both p.first and
                                             // p.second point to the
}                                            // insertion location for
                                             // the value searched for
```

```
VWIterPair p = equal_range(vw.begin(), vw.end(), w);
cout  << "There are " << distance(p.first, p.second)
      << " elements in vw equivalent to w.";
```

```
class Timestamp { ... };

bool operator<(const Timestamp& lhs,          // returns whether lhs
               const Timestamp& rhs);         // precedes rhs in time

vector<Timestamp> vt;                         // create vector, fill it with
...                                           // data, sort it so that older
sort(vt.begin(), vt.end());                   // times precede newer ones
```

```
Timestamp ageLimit;
...
vt.erase(vt.begin(), lower_bound(vt.begin(),          // eliminate from vt all
                                 vt.end(),            // objects that precede
                                 ageLimit));          // ageLimit's value
```

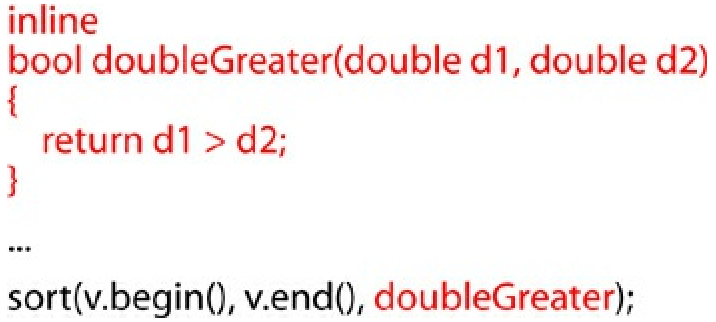
```
vt.erase(vt.begin(), upper_bound(vt.begin(),        // eliminate from vt all
                                 vt.end(),          // objects that precede
                                 ageLimit));        // or are equivalent
                                                    // to ageLimit's value
```

```
class Person {
public:
   ...
   const string& name() const;
   ...
};

struct PersonNameLess:
   public binary_function<Person, Person, bool> {        // see Item 40

   bool operator()(const Person& lhs, const Person& rhs) const
   {
      return lhs.name() < rhs.name();
   }

};

list<Person> lp;
...
lp.sort(PersonNameLess());                                // sort lp using
                                                          // PersonNameLess
```

```
Person newPerson;
...
lp.insert( upper_bound(lp.begin(),                // insert newPerson after
                       lp.end(),                  // the last object in lp
                       newPerson,                 // that precedes or is
                       PersonNameLess()),         // equivalent to
           newPerson);                            // newPerson
```

```
set<Widget> s;                  // create set, put data into it
...
Widget w;                       // w still holds the value to search for
...
if (s.count(w)) {
   ...                          // a value equivalent to w exists
} else {
   ...                          // no such value exists
}
```

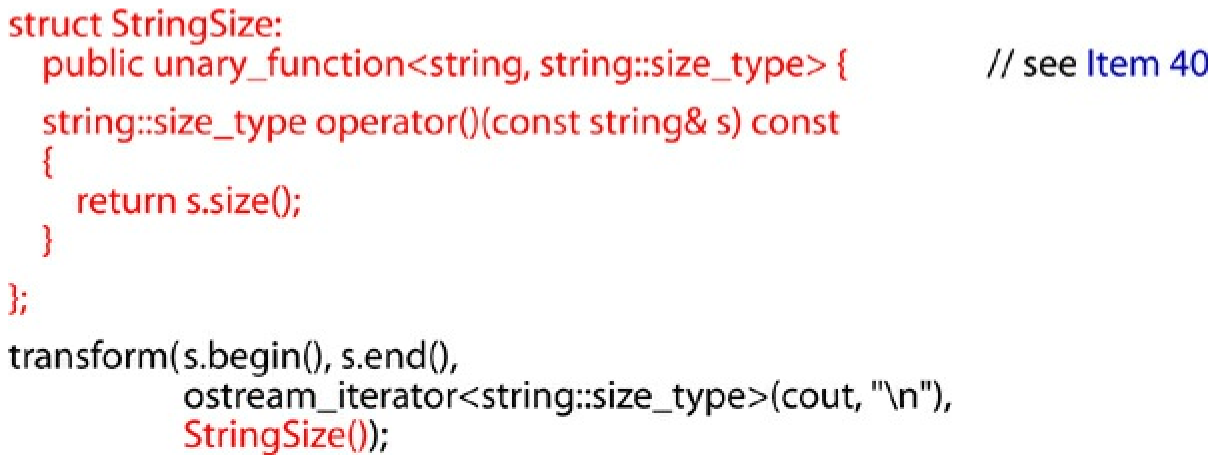
```
vector<double> v;
...
sort(v.begin(), v.end(), greater<double>());
```



```
inline
bool doubleGreater(double d1, double d2)
{
   return d1 > d2;
}

...

sort(v.begin(), v.end(), doubleGreater);
```

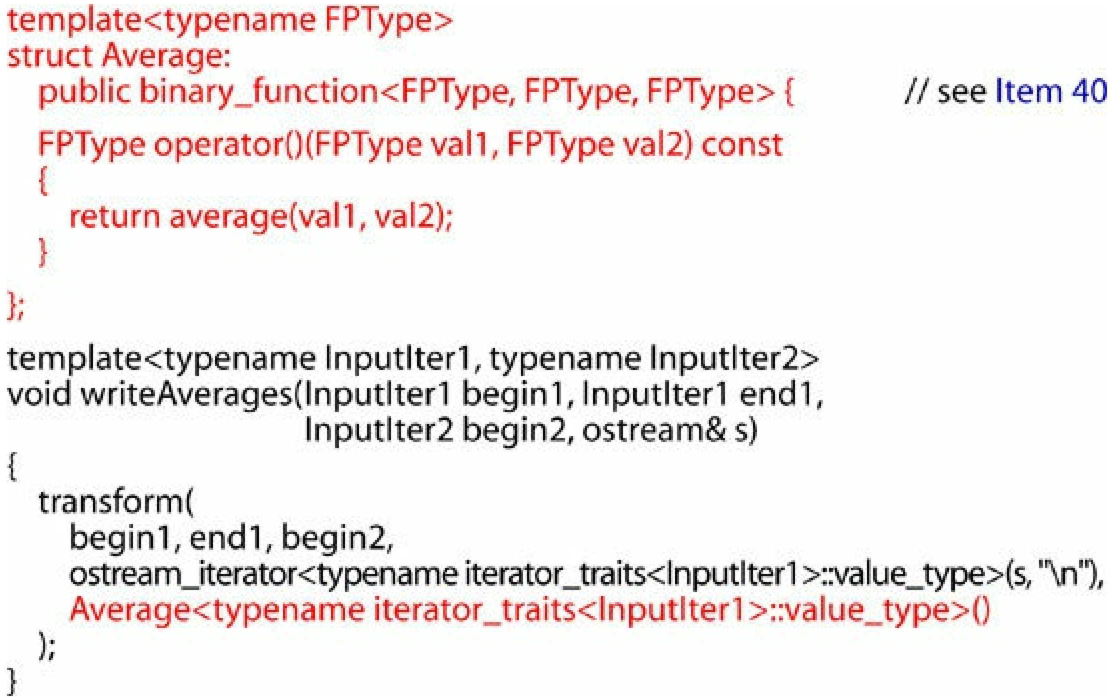
```
void sort( vector<double>::iterator first,          // beginning of range
           vector<double>::iterator last,           // end of range
           bool (*comp)(double, double));           // comparison function
```

```
set<string> s;
...
transform( s.begin(), s.end(),
           ostream_iterator<string::size_type>(cout, "\n"),
           mem_fun_ref(&string::size));
```



```
struct StringSize:
   public unary_function<string, string::size_type> {              // see Item 40

   string::size_type operator()(const string& s) const
   {
      return s.size();
   }

};

transform(s.begin(), s.end(),
               ostream_iterator<string::size_type>(cout, "\n"),
               StringSize());
```

```
template<typename FPType>                                   // return the average
FPType average(FPType val1, FPType val2)                    // of 2 floating point
{                                                           // numbers
   return (val1 + val2) / 2;
}

template< typename InputIter1,
          typename InputIter2>
void writeAverages( InputIter1 begin1,                      // write the pairwise
                    InputIter1 end1,                        // averages of 2
                    InputIter2 begin2,                      // sequences to a
                    ostream& s)                             // stream
{
   transform(
      begin1, end1, begin2,
      ostream_iterator<typename iterator_traits<InputIter1>::value_type>(s, "\n"),
      average<typename iterator_traits<InputIter1>::value_type>    // error?
   );
}
```



```
template<typename FPType>
struct Average:
  public binary_function<FPType, FPType, FPType> {          // see Item 40
  FPType operator()(FPType val1, FPType val2) const
  {
    return average(val1, val2);
  }
};
template<typename InputIter1, typename InputIter2>
void writeAverages(InputIter1 begin1, InputIter1 end1,
                   InputIter2 begin2, ostream& s)
{
  transform(
    begin1, end1, begin2,
    ostream_iterator<typename iterator_traits<InputIter1>::value_type>(s, "\n"),
    Average<typename iterator_traits<InputIter1>::value_type>()
  );
}
```

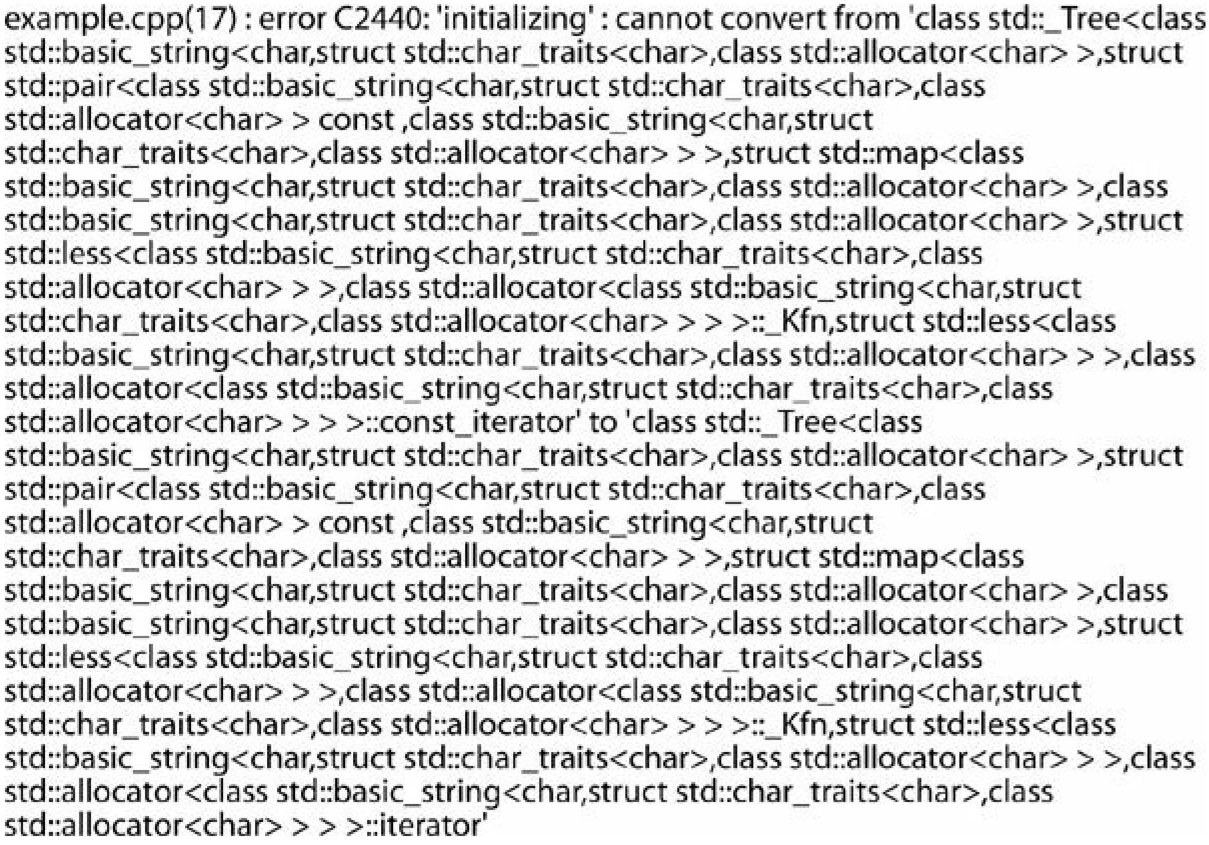
```
vector<int> v;
int x, y;

...

v.erase(
   remove_if(find_if(v.rbegin(), v.rend(),
                     bind2nd(greater_equal<int>(), y)).base(),
             v.end(),
             bind2nd(less<int>(), x)),
   v.end());
```

```
typedef vector<int>::iterator VecIntIter;

// Initialize rangeBegin to point to the element following the last
// occurrence of a value greater than or equal to y.  If there is no such
// value, initialize rangeBegin to v.begin().  If the last occurrence of the
// value is the last element in v, initialize rangeBegin to v.end().
VecIntIter rangeBegin = find_if(v.rbegin(), v.rend(),
                                   bind2nd(greater_equal<int>(), y)).base();

// from rangeBegin to v.end(), erase everything with a value less than x
v.erase(remove_if(rangeBegin, v.end(), bind2nd(less<int>(), x)), v.end());
```

```
v.erase(remove_if(find_if(v.rbegin(), v.rend(), something).base(),
                  v.end(),
                  something)),
        v.end());
```

```
class NiftyEmailProgram {
private:
   typedef map<string, string> NicknameMap;
   NicknameMap nicknames;                 // map from nicknames to
                                          // email addresses
public:
   ...
   void showEmailAddress(const string& nickname) const;
};
```

```
void NiftyEmailProgram::showEmailAddress(const string& nickname) const
{
   ...
   NicknameMap::iterator i = nicknames.find(nickname);
   if (i != nicknames.end()) ...
   ...
}
```



```
example.cpp(17) : error C2440: 'initializing' : cannot convert from 'class std::_Tree<class
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >,struct
std::pair<class std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> > const ,class std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> > >,struct std::map<class
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >,class
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >,struct
std::less<class std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> > >,class std::allocator<class std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> > > >::_Kfn,struct std::less<class
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > >,class
std::allocator<class std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> > > >::const_iterator' to 'class std::_Tree<class
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >,struct
std::pair<class std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> > const ,class std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> > >,struct std::map<class
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >,class
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >,struct
std::less<class std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> > >,class std::allocator<class std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> > > >::_Kfn,struct std::less<class
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > >,class
std::allocator<class std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> > > >::iterator'
```

```
example.cpp(17) : error C2440: 'initializing' : cannot convert from 'class
std::_Tree<class string,struct std::pair<class string const ,class string
>,struct std::map<class string,class string,struct std::less<class string
>,class std::allocator<class string > >::_Kfn,struct std::less<class string
>,class std::allocator<class string > >::const_iterator' to 'class
std::_Tree<class string,struct std::pair<class string const ,class string
>,struct std::map<class string,class string,struct std::less<class string
>,class std::allocator<class string > >::_Kfn,struct std::less<class string
>,class std::allocator<class string > >::iterator'
```

```
class NiftyEmailProgram {
private:
   typedef map<string, string> NicknameMap;
   NicknameMap nicknames;
public:
   ...
   void showEmailAddress(const string& nickname) const;
};
void NiftyEmailProgram::showEmailAddress(const string& nickname) const
{
   ...
   NicknameMap::iterator i = nicknames.find(nickname);
   if (i != nicknames.end()) ...
   ...
}
```

```
vector<string*> v;                                  // try to print a container
copy(v.begin(), v.end(),                            // of string* pointers as
        ostream_iterator<string>(cout, "\n"));      // string objects
```

```
list<int>::iterator i1, i2;             // pass bidirectional iterators to
sort(i1, i2);                           // an algorithm requiring random
                                        // access iterators
```

```
map<int, string> m;

...

// write all the map keys to cout
transform(m.begin(), m.end(),
          ostream_iterator<int>(cout, "\n"),
          select1st<map<int, string>::value_type>());

// create a vector and copy all the values in the map into it
vector<string> v;
transform(m.begin(), m.end(), back_inserter(v),
          select2nd<map<int, string>::value_type>());
```

```
int transmogrify(int x);                      // this function produces
                                              // some new value from x

vector<int> values;

...                                           // put data into values

vector<int> results;
transform( values.begin(), values.end(),      // this will attempt to
           results.end(),                     // write beyond the
           transmogrify);                     // end of results!
```

```
class Widget {
public:
   ...
   int readStream(istream& stream);                      // readStream takes
   ...                                                   // its parameter by
};                                                       // reference

vector<Widget*> vw;

...

for_each(                                                // most STL platforms
   vw.begin(), vw.end(),                                 // try to generate a
   bind2nd(mem_fun(&Widget::readStream), cin)            // reference to a
);                                                       // reference in this
                                                         // call; such code
                                                         // won't compile
```

```
struct lt_nocase
  : public std::binary_function<char, char, bool> {
   bool operator()(char x, char y) const {
     return  std::toupper(static_cast<unsigned char>(x)) <
             std::toupper(static_cast<unsigned char>(y));
   }
};
```

```
int main()
{
   const char* s1 = "GEW\334RZTRAMINER";
   const char* s2 = "gew\374rztraminer";
   printf("s1 = %s, s2 = %s\n", s1, s2);

   printf("s1 < s2: %s\n",
            std::lexicographical_compare(s1, s1 + 14, s2, s2 + 14, lt_nocase())
            ? "true" : "false");
}
```

```
struct lt_str_1
  : public std::binary_function<std::string, std::string, bool> {

  struct lt_char {
    const std::ctype<char>& ct;

    lt_char(const std::ctype<char>& c) : ct(c) {}

    bool operator()(char x, char y) const {
      return ct.toupper(x) < ct.toupper(y);
    }

  };

  std::locale loc;
  const std::ctype<char>& ct;

  lt_str_1(const std::locale& L = std::locale::classic())
    : loc(L), ct(std::use_facet<std::ctype<char> >(loc)) {}

  bool operator()(const std::string& x, const std::string& y) const {
    return std::lexicographical_compare( x.begin(), x.end(),
                                         y.begin(), y.end(),
                                         lt_char(ct));
  }

};
```

```
struct lt_str_2 :
    public std::binary_function<std::string, std::string, bool> {

    struct lt_char {
        const char* tab;

        lt_char(const char* t) : tab(t) {}

        bool operator()(char x, char y) const {
            return tab[x - CHAR_MIN] < tab[y - CHAR_MIN];
        }

    };

    char tab[CHAR_MAX - CHAR_MIN + 1];

    lt_str_2(const std::locale& L = std::locale::classic()) {
        const std::ctype<char>& ct = std::use_facet<std::ctype<char> >(L);

        for (int i = CHAR_MIN; i <= CHAR_MAX; ++i)
            tab[i - CHAR_MIN] = (char) i;

        ct.toupper(tab, tab + (CHAR_MAX - CHAR_MIN + 1));
    }

    bool operator()(const std::string& x, const std::string& y) const {
        return std::lexicographical_compare(x.begin(), x.end(),
                                            y.begin(), y.end(),
                                            lt_char(tab));
    }

};
```

```
vector<Widget> vw1, vw2;
...
vw1.insert(vw1.end(), vw2.begin(), vw2.end());     // append to vw1 a copy
                                                   // of the Widgets in vw2
```

```
vector<Widget> vw;
deque<Widget> dw;

...

vw.insert(vw.end(), dw.begin(), dw.end());            // append to vw a copy
                                                      // of the Widgets in dw
```

```
vector<Widget> vw;
...
list<Widget> lw;
...
vw.insert(vw.begin(), lw.begin(), lw.end());          // prepend to vw a copy
                                                      // of the Widgets in lw
set<Widget> sw;
...
vw.insert(vw.begin(), sw.begin(), sw.end());          // prepend to vw a copy
                                                      // of the Widgets in sw
template<typename T,                                  // template for a custom
         typename Allocator = allocator<T> >          // STL-compatible
class SpecialContainer { ... };                       // container
SpecialContainer<Widget> scw;
...
vw.insert(vw.end(), scw.begin(), scw.end());          // append to vw a copy
                                                      // of the Widgets in scw
```

```
template <class T, class Allocator = allocator<T> >
class vector {
public:

    ...

    template <class InputIterator>
    void insert(iterator position, InputIterator first, InputIterator last);

    ...

};
```

```
void insert( iterator position,                  // "iterator" is the
             iterator first, iterator last);     // container's iterator type
```

```
istream_iterator<Widget> begin(cin), end;      // create begin and end
                                               // iterators for reading
                                               // Widgets from cin
                                               // (see Item 6)
vector<Widget> vw(begin, end);                 // read cin's Widgets into vw
                                               // (again, see Item 6); won't
                                               // compile with MSVC4–6
list<Widget> lw;
...
lw.assign(vw.rbegin(), vw.rend());             // assign vw's contents to lw
                                               // (in reverse order); won't
                                               // compile with MSVC4–6
SpecialContainer<Widget> scw;
...
scw.insert(scw.end(), lw.begin(), lw.end());   // insert at the end of scw a
                                               // copy of the Widgets in lw;
                                               // won't compile with
MSVC4–6
```

```
vector<Widget> vw(begin, end);                    // rejected by the MSVC4-6
                                                  // STL implementation
list<Widget> lw;
...
lw.assign(vw.rbegin(), vw.rend());                //also rejected
SpecialContainer<Widget> scw;
...
scw.insert(scw.end(), lw.begin(), lw.end());      // ditto
```

```
istream_iterator<Widget> begin(cin), end;

vector<Widget> vw;                                   // default-construct vw;
copy(begin, end, back_inserter(vw));                 // then copy the
                                                     // Widgets in cin into it


list<Widget> lw;
...
lw.clear();                                          // eliminate lw's old
copy(vw.rbegin(), vw.rend(), back_inserter(lw));     // Widgets; copy over
                                                     // vw's Widgets (in
                                                     // reverse order)

SpecialContainer<Widget> scw;
...
copy( lw.begin(), lw.end(),                          // copy lw's Widgets to
      inserter(scw, scw.end()));                     // the end of scw
```